React

Author: Pavel Šeda

Materials supported by online courses (e.g., Udemy: React - The Complete Guide), other publicly available online courses, books, tutorials, etc.

Audience

- Beginners to React, who have logical, and analytical problem-solving skills and who want to begin with learning React for the purpose of building front-end web applications

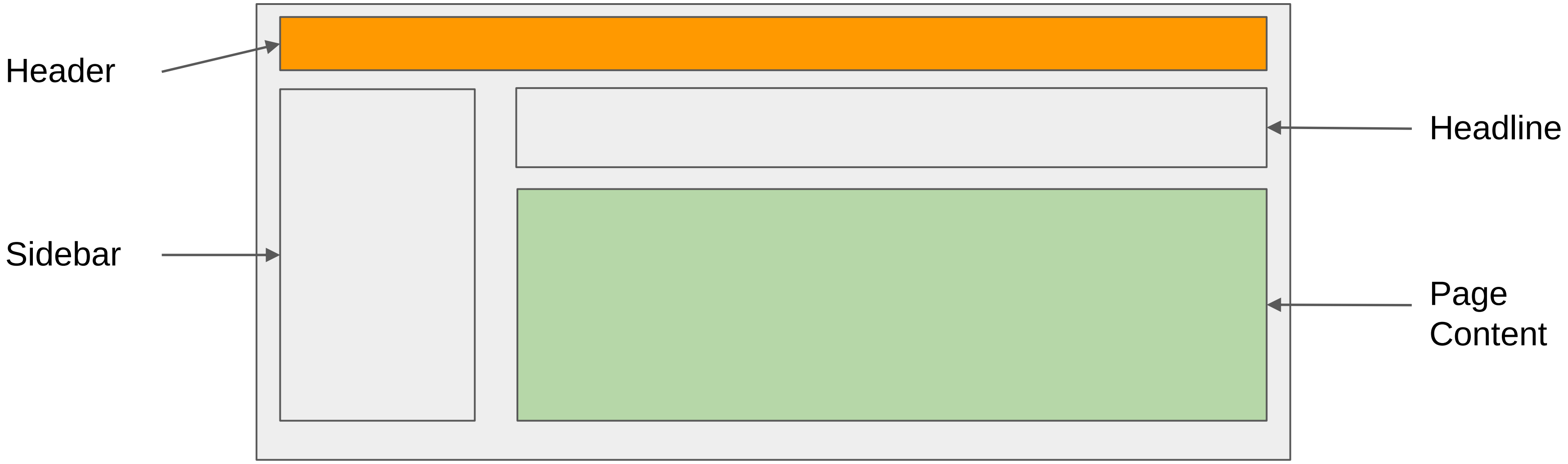
Course Objectives

- Introduction and comparison of React with others
- The basics
    - Basic concepts/features and syntax
    - Debugging
    - Styling Components
- Components
- HTTP requests
- Routing
- Forms & validation
- Redux
- Deployment

Section: Introduction and comparison of React with others

Introduction to React

- What is React?
  - “A **Javascript library** for building User Interfaces”
  - Originally developed and released by **Facebook in 2013**
- React uses components to separate page with predefined components



Introduction to React

- Declarative
  - React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes
  - Declarative views make your code more predictable and easier to debug
- Component-Based
  - Build encapsulated components that manage their own state, then compose them to make complex UIs
  - Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep state out of the DOM
- Learn Once, Write Anywhere
  - We don’t make assumptions about the rest of your technology stack, so you can develop new features in React without rewriting existing code
  - React can also render on the server using Node and power mobile apps using React Native

Why React?

- UI State becomes difficult to handle with Vanilla JavaScript
  - In complex web pages it is almost impossible to manage such situations
- Huge ecosystem, active community, high performance
  - Support from Facebook and other companies
- Focus on business logic

React Alternatives

Nowadays there are basically the three mainstream approaches to build a web page:

1. React (backed by Facebook)
   a. Only view library
   b. Requires additional libraries to make it fully work
   c. Originally built-for high traffic applications
   d. Used by: Whatsapp, Uber, Instagram, Paypal, Glassdoor, BBC, Facebook, ...
2. Angular (backed by Google)
   a. Initially released in 2010 as AngularJS but in 2016 totally reworked and renamed to Angular
   b. Typescript based JavaScript framework
   c. Fully-fledged framework
   d. Used by: Gmail, HBO, General Motors
3. Vue (backed by Community)
   a. Initially released in 2014 from ex-Google developer Evan You
   b. The lowest demand on the job markets (LinkedIn, Indeed) from these alternatives
   c. Used by: GitLab, Alibaba, Udemy

React Alternatives

What about jQuery?

- It is not an alternative since it is **only about traversing the DOM**
- The previously mentioned approaches (React, Angular, Vue) are more focused on a logic instead of a BOOM how effect
- The jQuery applications are **hardly maintainable** when the project gets bigger

React vs Angular Comparison

React	**Factor**	**Angular**
JavaScript	Programming Language	Typescript
Moderate	Learning Curve	Steep
Component Based	Architecture	Component Based
Relatively Small	App Size	Relatively Small
Virtual DOM	DOM	Real DOM
Client/Server Side	Rendering	Client/Server Side
JSX+JS (ES5 and Beyond)	Template	HTML+Typescript
Unidirectional (one-way)	Data Binding	Bidirectional (two-way)
Medium	Abstraction	Strong

Section: The Basics

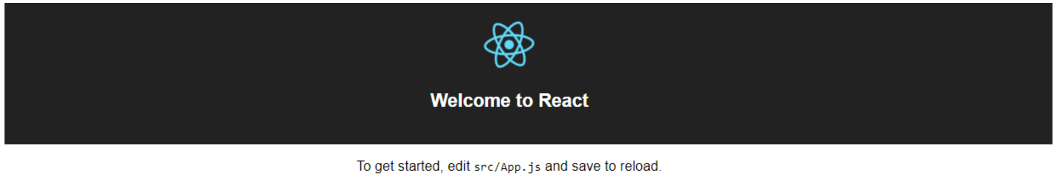
Create React App

1. Download and Install NodeJS - https://nodejs.org/en/
    a. This is required to use **npm dependency manager** and to use the **development server**
2. Cmd: “npm install create-react-app -g”
    a. The -g command means that it will be accessible globally
    b. On Linux it is required to prefix that command using sudo
3. Go to working directory where the application should be
4. Cmd: “create-react-app seda-react-guide --scripts-version 1.1.5”
    a. The --scripts-version 1.1.5 means that it will create a project structure based on that version -- this is not a required options
5. Start the applications:
    a. Cmd: “cd seda-react-guide”
    b. Cmd: “npm start”

Create React App

After the previous steps visit the page: http://localhost:3000 on your browser

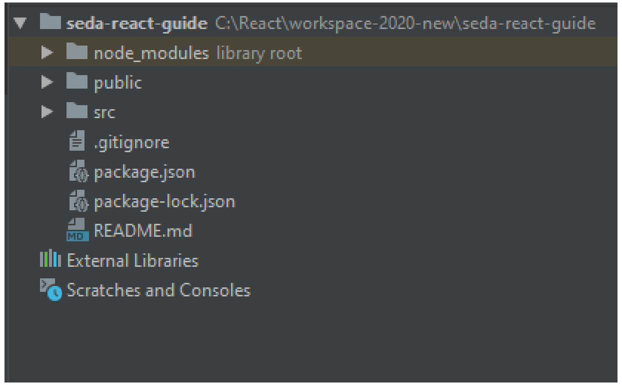
You should see the following page:


Welcome to React

To get started, edit `src/App.js` and save to reload.

Create React App

Now the project structure is as follows:



```
▼ seda-react-guide C:\React\workspace-2020-new\seda-react-guide
  ▶ node_modules library root
  ▶ public
  ▶ src
    .gitignore
    package.json
    package-lock.json
    README.md
  External Libraries
  Scratches and Consoles
```

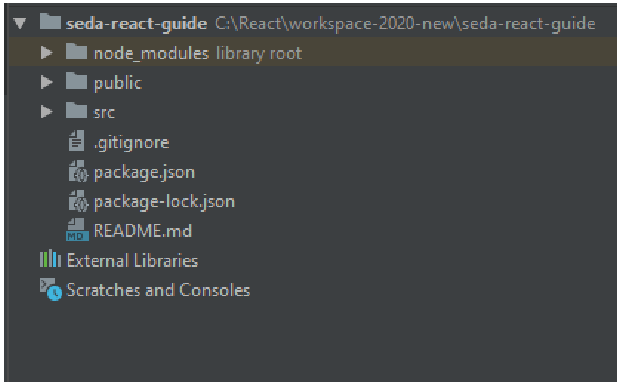
Where:

- package-lock.json
  - Describe the dependency tree at a given moment.
  - Provide the ability to go back in time to a specific version of the dependency tree.
  - Optimize the installation process by allowing the existence of NPM to skip existing items.

- package.json
  - Is used for more than dependencies - like defining project properties, description, author & license information, scripts, etc.
  - The package-lock.json is solely used to lock dependencies to a specific version number

Create React App

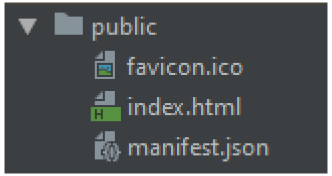
Now the project structure is as follows:



```
▼ seda-react-guide C:\React\workspace-2020-new\seda-react-guide
  ▶ node_modules library root
  ▶ public
  ▶ src
    .gitignore
    package.json
    package-lock.json
    README.md
  External Libraries
  Scratches and Consoles
```

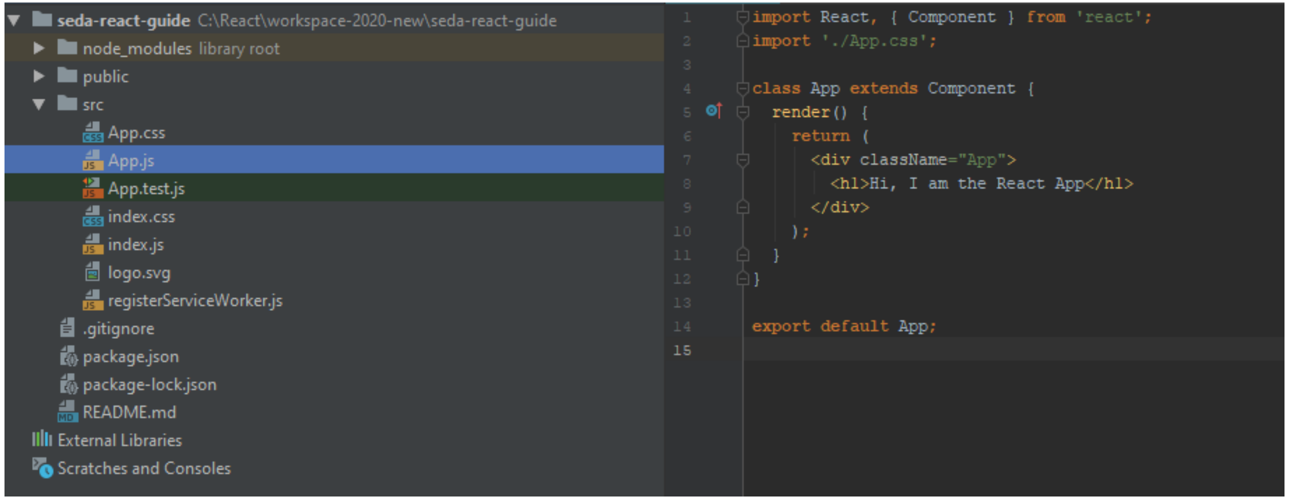
Where:

- node_modules folder
  - Holds all the dependencies
  - You should not add anything to that folder
- src folder
  - Contains the couple of files that actually represents our React application
  - Index.js renders our app root component
- public folder
  - This holds the files that we can edit
  - Contains **index.html** page it is the only one .html page in the project
  - **manifest.json** is used to define some metadata of the application

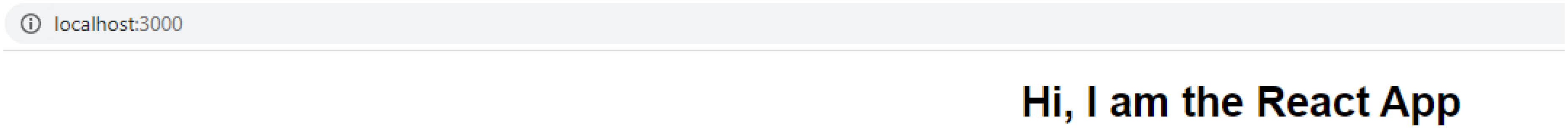


```
▼ public
    favicon.ico
    index.html
    manifest.json
```

Create React App - Modifying Your First App



```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <h1>Hi, I am the React App</h1>
      </div>
    );
  }
}

export default App;
```

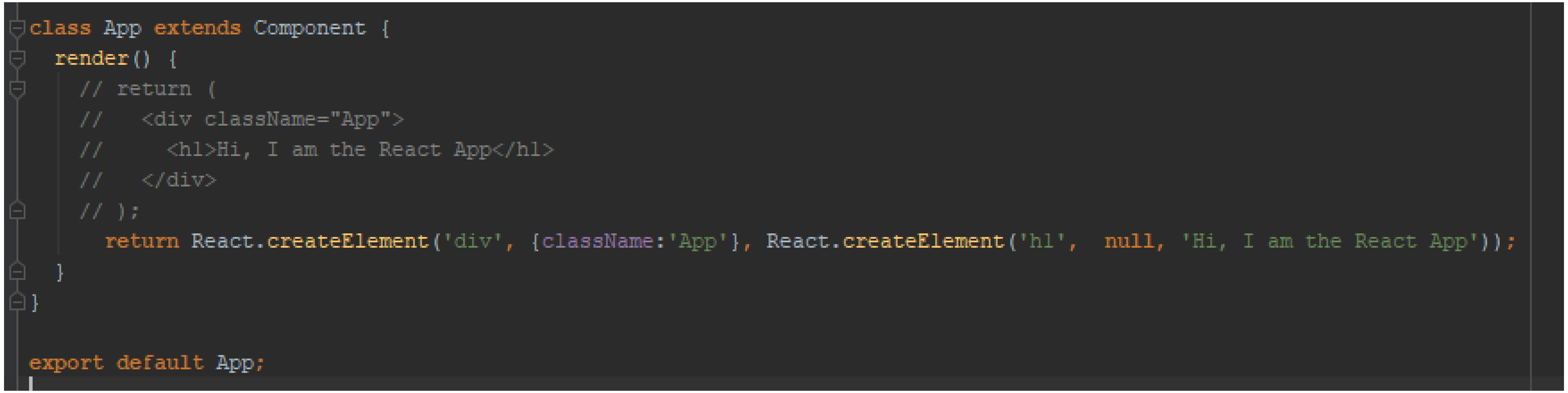


localhost:3000

Hi, I am the React App

React JSX

- React components are usually written in HTML-like structure (JSX) that is internally compiled to React.createElement(...)
    a. Here, you can see the same code written in a different way (it is exact equivalent)



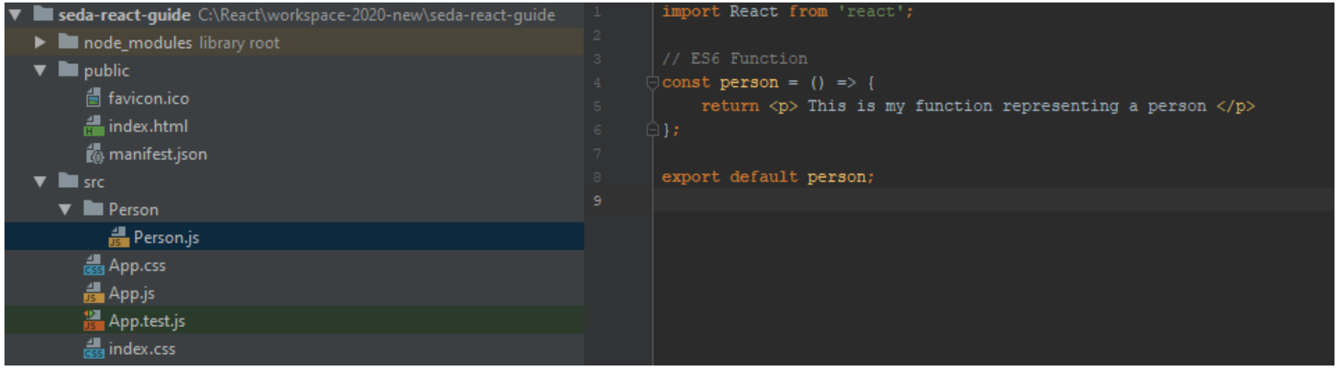
```
class App extends Component {
  render() {
    // return (
    //   <div className="App">
    //     <h1>Hi, I am the React App</h1>
    //   </div>
    // );
      return React.createElement('div', {className:'App'}, React.createElement('h1',  null, 'Hi, I am the React App'));
  }
}

export default App;
```

JSX Restrictions

- <div class=”App”> the world class can not be used since it is reserved word in JavaScript, instead we need to use <div className=”App”> (internally, it is translated to <div class=”App”>
- JSX expressions must have one root element
  - I cannot create something like this: return (<div>...</div><h1></h1>)
- Pay special attention to these since it looks like HTML, but it is NOT! It’s JSX

Functional Components

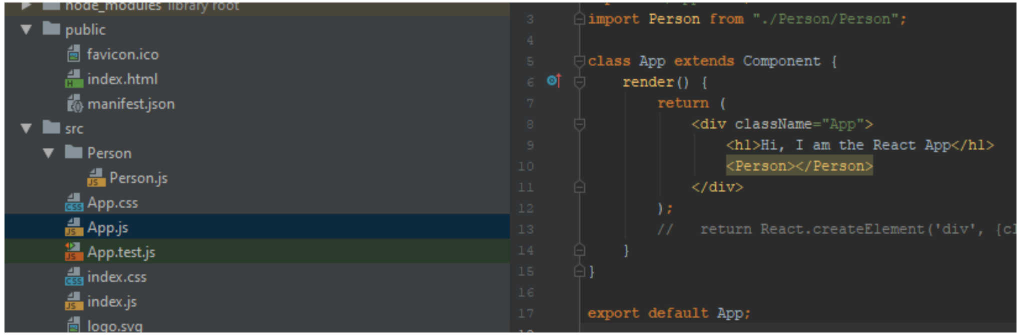
- Creating a functional component in ES6



```
▼ seda-react-guide C:\React\workspace-2020-new\seda-react-guide
  ▶ node_modules library root
  ▼ public
      favicon.ico
      index.html
      manifest.json
  ▼ src
    ▼ Person
        Person.js
      App.css
      App.js
      App.test.js
      index.css
```

```
import React from 'react';

// ES6 Function
const person = () => {
    return <p> This is my function representing a person </p>
};

export default person;
```

- Using this component in another part of the project
  - Check that part (<Person/>) in Developer Console, the tag would not be as <Person> but as <p>



```
  ▶ node_modules library root
  ▼ public
      favicon.ico
      index.html
      manifest.json
  ▼ src
    ▼ Person
        Person.js
      App.css
      App.js
      App.test.js
      index.css
      index.js
      logo.svg
```

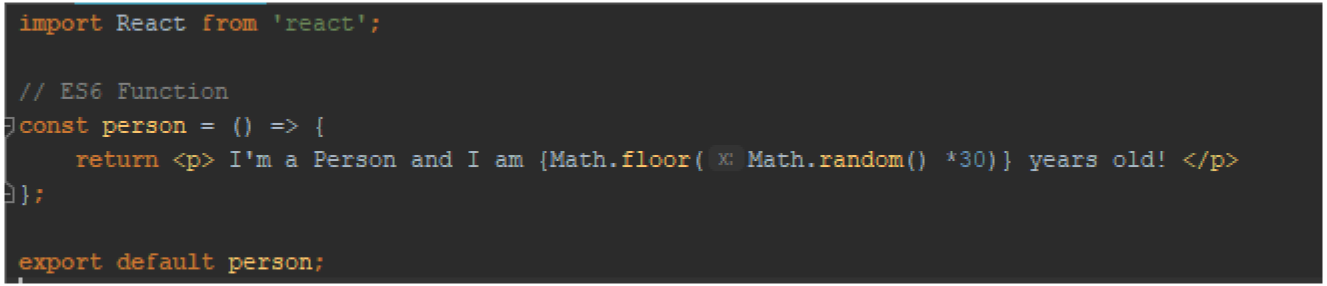
```
import Person from "./Person/Person";

class App extends Component {
    render() {
        return (
            <div className="App">
                <h1>Hi, I am the React App</h1>
                <Person></Person>
            </div>
        );
        //   return React.createElement('div', {cl
    }
}

export default App;
```

Components & JSX Cheat Sheet

- Components are the core building block of React apps
  - React is just a library for creating components in its core
- A typical React app therefore could be depicted as a component tree
  - It has one root component ("App") and then an potentially infinite amount of nested child components
- Each component needs to return/render some JSX code
  - It defines which HTML code React should render to the real DOM in the end

Dynamic Content in Functional Component

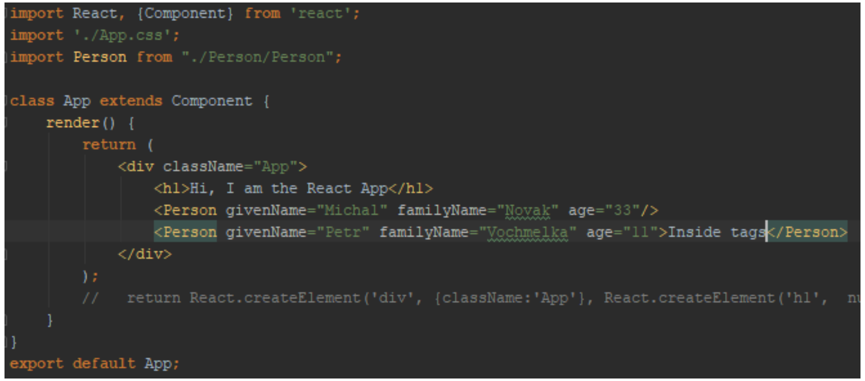
- Calling a JavaScript functions inside components
- Making a content dynamic



```
import React from 'react';

// ES6 Function
const person = () => {
    return <p> I'm a Person and I am {Math.floor( Math.random() *30)} years old! </p>
};

export default person;
```

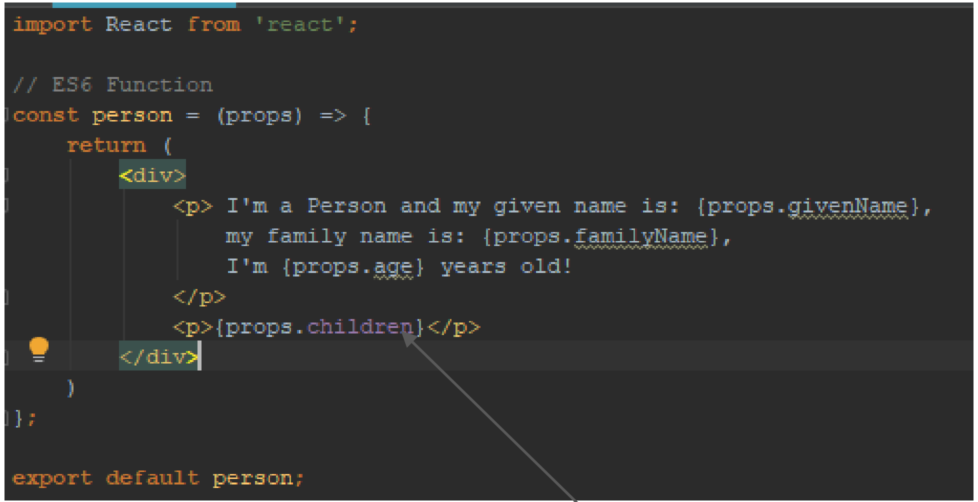
Using a Properties to Pass Args to the Components

- Pass arguments for Person Object



```
import React, {Component} from 'react';
import './App.css';
import Person from "./Person/Person";

class App extends Component {
    render() {
        return (
            <div className="App">
                <h1>Hi, I am the React App</h1>
                <Person givenName="Michal" familyName="Novak" age="33"/>
                <Person givenName="Petr" familyName="Vochmelka" age="11">Inside tags</Person>
            </div>
        );
        //   return React.createElement('div', {className:'App'}, React.createElement('h1',  nu
    }
}
export default App;
```

- Modify our component to access properties

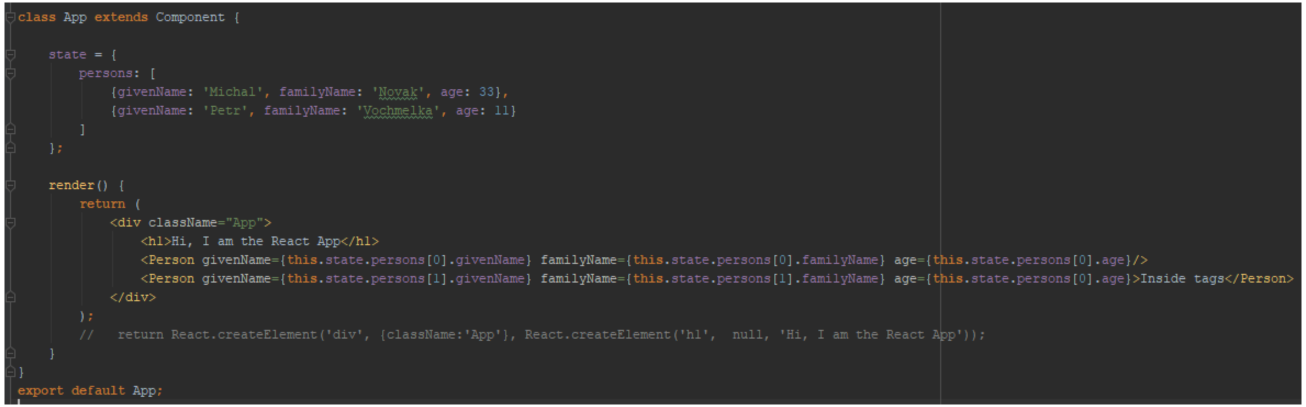


```
import React from 'react';

// ES6 Function
const person = (props) => {
    return (
        <div>
            <p> I'm a Person and my given name is: {props.givenName},
                my family name is: {props.familyName},
                I'm {props.age} years old!
            </p>
            <p>{props.children}</p>
        </div>
    )
};

export default person;
```

Content between tags

Understanding and Using State (class-based React Components)

- Defining the state in class-based React Component:



```
class App extends Component {

    state = {
        persons: [
            {givenName: 'Michal', familyName: 'Novak', age: 33},
            {givenName: 'Petr', familyName: 'Vochmelka', age: 11}
        ]
    };

    render() {
        return (
            <div className="App">
                <h1>Hi, I am the React App</h1>
                <Person givenName={this.state.persons[0].givenName} familyName={this.state.persons[0].familyName} age={this.state.persons[0].age}/>
                <Person givenName={this.state.persons[1].givenName} familyName={this.state.persons[1].familyName} age={this.state.persons[0].age}>Inside tags</Person>
            </div>
        );
        //   return React.createElement('div', {className:'App'}, React.createElement('h1',  null, 'Hi, I am the React App'));
    }
}
export default App;
```

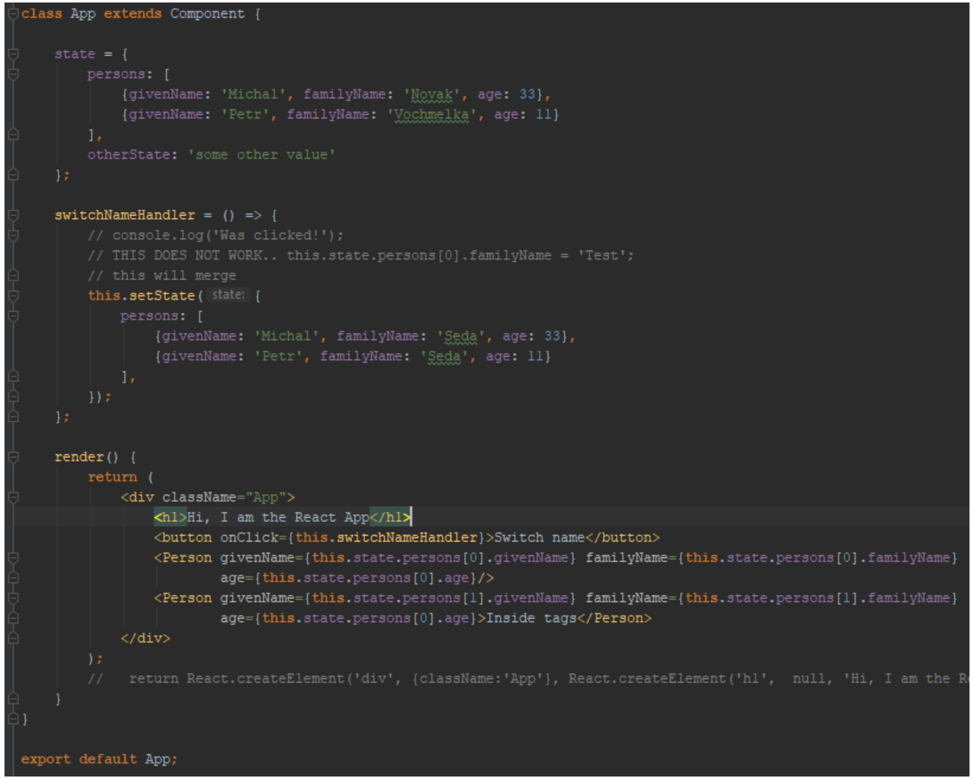
- If we change state in that component it will lead React to rerender/update our DOM

Props & State

- **props** and **state** are CORE concepts of React
- Changes in **props** and/or **state** triggers React to re-render your components

Handling Events with Methods

- The state of the component have to be changed with setState method
- The setState method is called asynchronously => It may take some time to re-render the new state



```
class App extends Component {

    state = {
        persons: [
            {givenName: 'Michal', familyName: 'Novak', age: 33},
            {givenName: 'Petr', familyName: 'Vochmelka', age: 11}
        ],
        otherState: 'some other value'
    };

    switchNameHandler = () => {
        // console.log('Was clicked!');
        // THIS DOES NOT WORK.. this.state.persons[0].familyName = 'Test';
        // this will merge
        this.setState( state: {
            persons: [
                {givenName: 'Michal', familyName: 'Seda', age: 33},
                {givenName: 'Petr', familyName: 'Seda', age: 11}
            ],
        });
    };

    render() {
        return (
            <div className="App">
                <h1>Hi, I am the React App</h1>
                <button onClick={this.switchNameHandler}>Switch name</button>
                <Person givenName={this.state.persons[0].givenName} familyName={this.state.persons[0].familyName}
                        age={this.state.persons[0].age}/>
                <Person givenName={this.state.persons[1].givenName} familyName={this.state.persons[1].familyName}
                        age={this.state.persons[0].age}>Inside tags</Person>
            </div>
        );
        //   return React.createElement('div', {className:'App'}, React.createElement('h1',  null, 'Hi, I am the R
    }
}

export default App;
```

Why do not access directly this.state

- Consider the following: ‘this.state.persons=”Pavel”’ ?
  - This does not work since React wants to update the state in an asynchronous manner
  - This is the reason so that React does not support this directly assigned variable
  - Also, consider that with updating state using the setState method it allows React to do a piece of “black magic” behind the scenes

React Supported Events

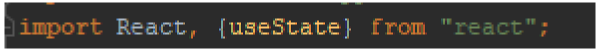
- React supports a wide variety of events that can be listened
- For the full list of supported events, please visit the following site:
  - https://reactjs.org/docs/events.html#supported-events

State In Functional Components

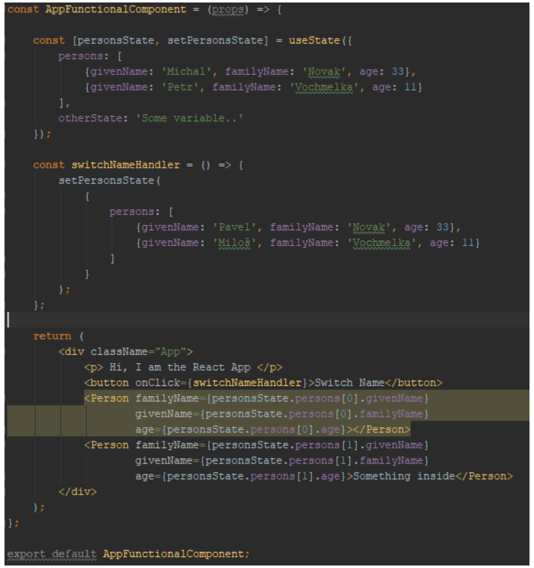
- Since React 16.8 there is a possibility to add state to functional components using the React Hooks
- Previously it was possible only in class-based components
- The class-based components to manage the state is an established way

State In Functional Components

- This is done by hooks, especially the **useState** hooks
- First you need to import the {useState} instead of {Component} as follows:



```
import React, {useState} from "react";
```

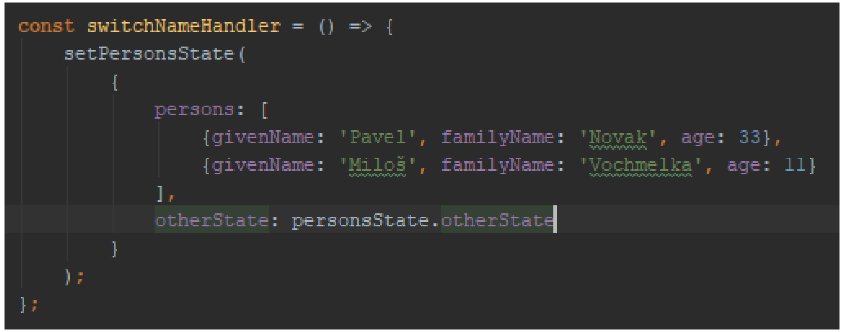


```
const AppFunctionalComponent = (props) => {

    const [personsState, setPersonsState] = useState({
        persons: [
            {givenName: 'Michal', familyName: 'Novak', age: 33},
            {givenName: 'Petr', familyName: 'Vochmelka', age: 11}
        ],
        otherState: 'Some variable..'
    });

    const switchNameHandler = () => {
        setPersonsState(
            {
                persons: [
                    {givenName: 'Pavel', familyName: 'Novak', age: 33},
                    {givenName: 'Miloš', familyName: 'Vochmelka', age: 11}
                ]
            }
        );
    };

    return (
        <div className="App">
            <p> Hi, I am the React App </p>
            <button onClick={switchNameHandler}>Switch Name</button>
            <Person familyName={personsState.persons[0].givenName}
                    givenName={personsState.persons[0].familyName}
                    age={personsState.persons[0].age}></Person>
            <Person familyName={personsState.persons[1].givenName}
                    givenName={personsState.persons[1].familyName}
                    age={personsState.persons[1].age}>Something inside</Person>
        </div>
    );
};

export default AppFunctionalComponent;
```

- Please, see that in the functional component we do not have **.this.state..** Call since we are not in the class-based components so **.this** is not allowed here
- The first argument in useState is for getting the current state (**personsState)** and the second is for setting up the state (**setPersonsState**)

State In Functional Components - Important Notes

- When you call the update on the functional component state
- It does not merge! As in the case of class-based components the current state with the new state



```
const switchNameHandler = () => {
    setPersonsState(
        {
            persons: [
                {givenName: 'Pavel', familyName: 'Novak', age: 33},
                {givenName: 'Miloš', familyName: 'Vochmelka', age: 11}
            ],
            otherState: personsState.otherState
        }
    );
};
```

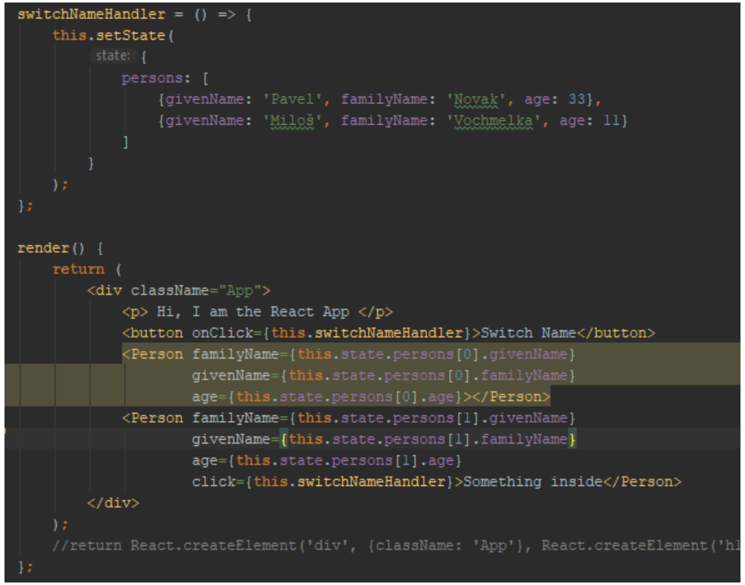
- **It replaces the new state at all**
- So you have to add all the old data into the new state setting
    - E.g., here using personsState.otherState

Stateless vs Stateful Components

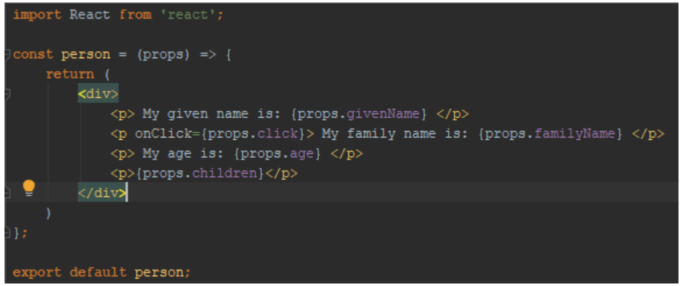
- Stateless
  - Presentational components (dump components)
  - These returns a data in a predefined expected structure
- Stateful
  - Sometimes called container components (smart components)
  - These usually contains plethora of stateless components
  - Manage the state of the application
- For the detailed comparison of stateless vs stateful components visit the following site:
  - https://programmingwithmosh.com/javascript/stateful-stateless-components-react/

Passing Method Reference Between Components

- We can pass method reference as here, we add a name for it as a click and in the stateless component (person) we just use this onClick event and access the method reference that is hidden by props.click property



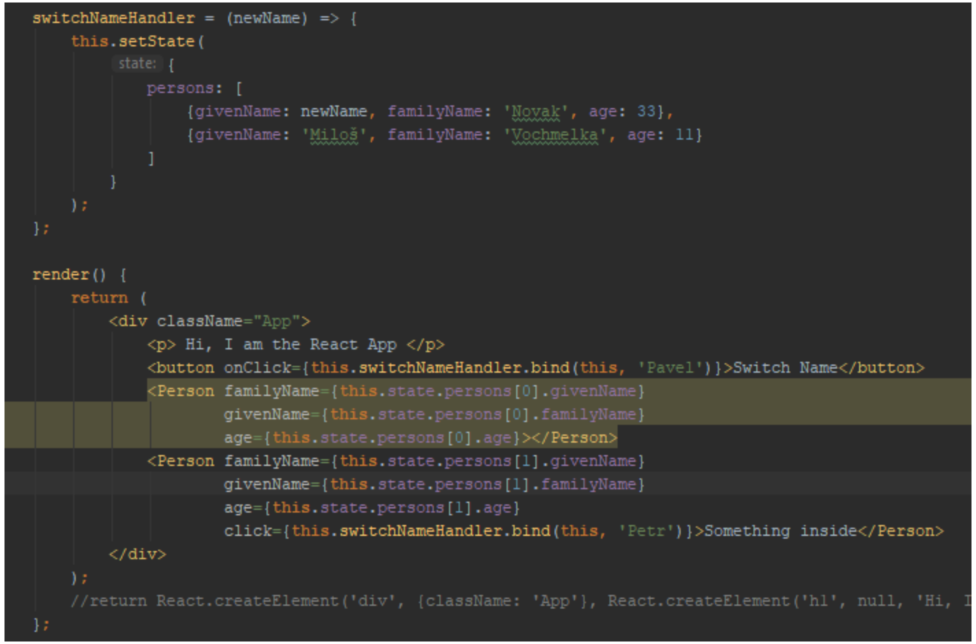
```
switchNameHandler = () => {
    this.setState(
        state: {
            persons: [
                {givenName: 'Pavel', familyName: 'Novak', age: 33},
                {givenName: 'Miloš', familyName: 'Vochmelka', age: 11}
            ]
        }
    );
};

render() {
    return (
        <div className="App">
            <p> Hi, I am the React App </p>
            <button onClick={this.switchNameHandler}>Switch Name</button>
            <Person familyName={this.state.persons[0].givenName}
                    givenName={this.state.persons[0].familyName}
                    age={this.state.persons[0].age}></Person>
            <Person familyName={this.state.persons[1].givenName}
                    givenName={this.state.persons[1].familyName}
                    age={this.state.persons[1].age}
                    click={this.switchNameHandler}>Something inside</Person>
        </div>
    );
    //return React.createElement('div', {className: 'App'}, React.createElement('h1
};
```



```
import React from 'react';

const person = (props) => {
    return (
        <div>
            <p> My given name is: {props.givenName} </p>
            <p onClick={props.click}> My family name is: {props.familyName} </p>
            <p> My age is: {props.age} </p>
            <p>{props.children}</p>
        </div>
    )
};

export default person;
```

Passing Method Reference Between Components

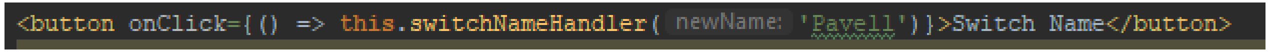
- Using binding to change the component state based on the place where I had clicked



```
switchNameHandler = (newName) => {
    this.setState(
        state: {
            persons: [
                {givenName: newName, familyName: 'Novak', age: 33},
                {givenName: 'Miloš', familyName: 'Vochmelka', age: 11}
            ]
        }
    );
};

render() {
    return (
        <div className="App">
            <p> Hi, I am the React App </p>
            <button onClick={this.switchNameHandler.bind(this, 'Pavel')}>Switch Name</button>
            <Person familyName={this.state.persons[0].givenName}
                    givenName={this.state.persons[0].familyName}
                    age={this.state.persons[0].age}></Person>
            <Person familyName={this.state.persons[1].givenName}
                    givenName={this.state.persons[1].familyName}
                    age={this.state.persons[1].age}
                    click={this.switchNameHandler.bind(this, 'Petr')}>Something inside</Person>
        </div>
    );
    //return React.createElement('div', {className: 'App'}, React.createElement('h1', null, 'Hi, I
};
```

Passing Method Reference Between Components

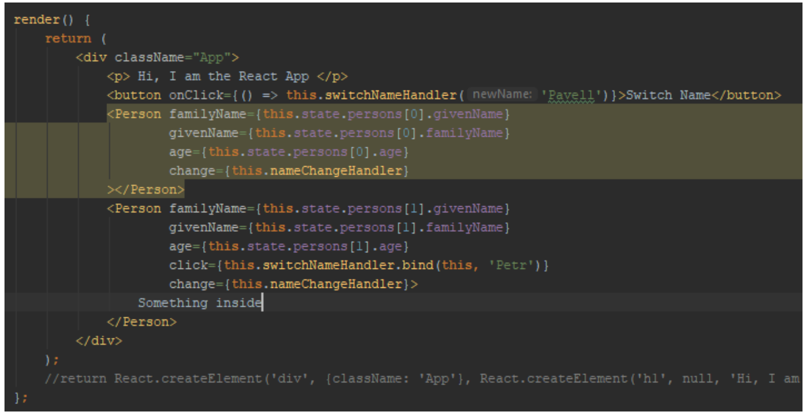
- Using binding to change the component state based on the place where I had clicked
- Using anonymous functions (please consider that there might be performance drawbacks



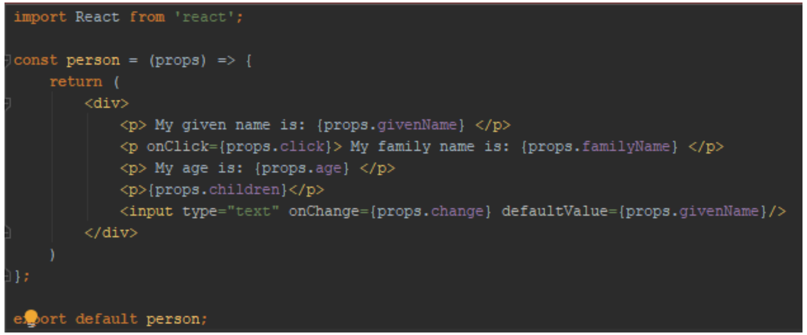
```
<button onClick={() => this.switchNameHandler( newName: 'Pavell')}>Switch Name</button>
```

React Two Way Binding

- Two way data binding can be achieved using events
  - We will pass a method reference that as an input receives an event
  - In the stateless component we assign it, e.g., with the event onChange for input element



```
render() {
    return (
        <div className="App">
            <p> Hi, I am the React App </p>
            <button onClick={() => this.switchNameHandler( newName: 'Pavell')}>Switch Name</button>
            <Person familyName={this.state.persons[0].givenName}
                    givenName={this.state.persons[0].familyName}
                    age={this.state.persons[0].age}
                    change={this.nameChangeHandler}
            ></Person>
            <Person familyName={this.state.persons[1].givenName}
                    givenName={this.state.persons[1].familyName}
                    age={this.state.persons[1].age}
                    click={this.switchNameHandler.bind(this, 'Petr')}
                    change={this.nameChangeHandler}>
                Something inside
            </Person>
        </div>
    );
    //return React.createElement('div', {className: 'App'}, React.createElement('h1', null, 'Hi, I am
};
```



```
import React from 'react';

const person = (props) => {
    return (
        <div>
            <p> My given name is: {props.givenName} </p>
            <p onClick={props.click}> My family name is: {props.familyName} </p>
            <p> My age is: {props.age} </p>
            <p>{props.children}</p>
            <input type="text" onChange={props.change} defaultValue={props.givenName}/>
        </div>
    )
};

export default person;
```

Tasks

1. Create TWO new components: PersonInput and PersonOutput
2. PersonInput should hold an input element, PersonOutput two paragraphs
3. Output multiple PersonOutput components in the App component (any paragraph texts of your choice)
4. Pass a username (of your choice) to PersonOutput via props and display it there
5. Add state to the App component (=> the username) and pass the username to the PersonOutput component
6. Add a method to manipulate the state (=> an event-handler method)
7. Ensure that the new input entered by the person overwrites the old username passed to PersonOutput

Section: Material Design

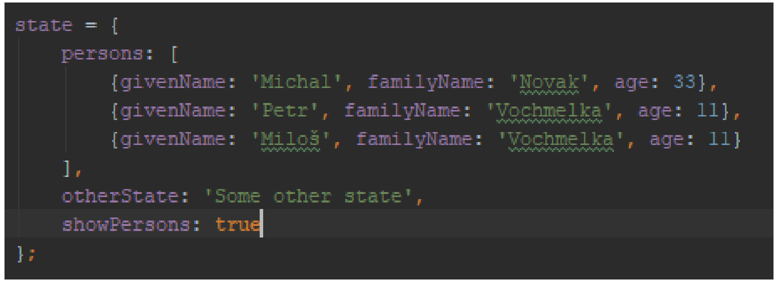
Material Design -- Intro

- Install material design into your project “npm install @material-ui/core”
  - “npm install @material-ui/core --save”
- Its as easy as it seems
- https://material-ui.com/
- https://material-ui.com/getting-started/installation/
- Go through the Component List
- Include necessary components to your project

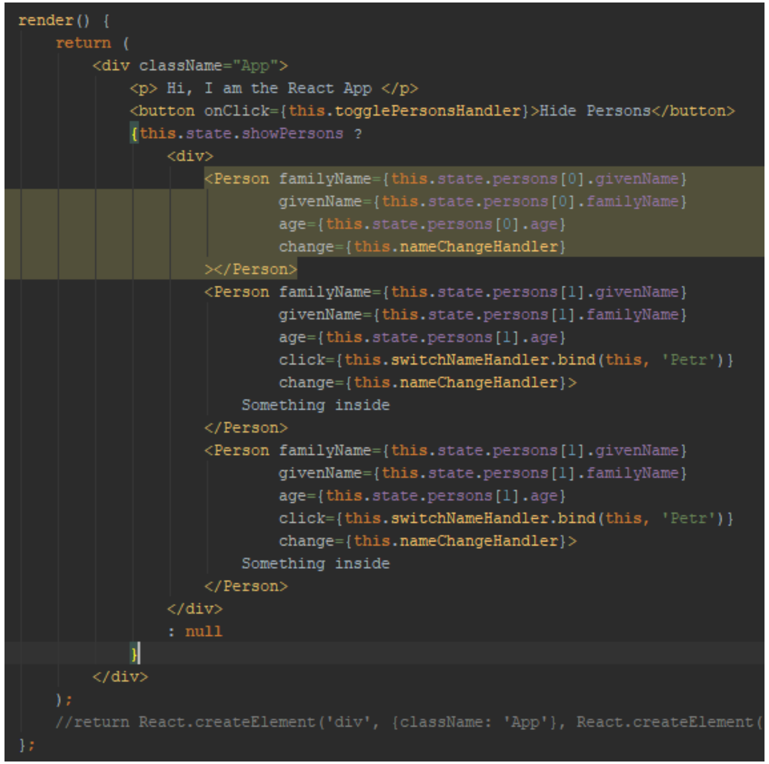
Section: Lists and Conditionals

IF Conditionals

- Consider that you want to show persons conditionally
  - E.g., add another field to the state object
  - The IF condition (ternary operator) is shown in the picture on the right



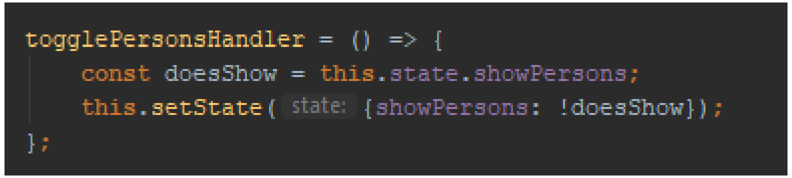
```
state = {
    persons: [
        {givenName: 'Michal', familyName: 'Novak', age: 33},
        {givenName: 'Petr', familyName: 'Vochmelka', age: 11},
        {givenName: 'Miloš', familyName: 'Vochmelka', age: 11}
    ],
    otherState: 'Some other state',
    showPersons: true
};
```



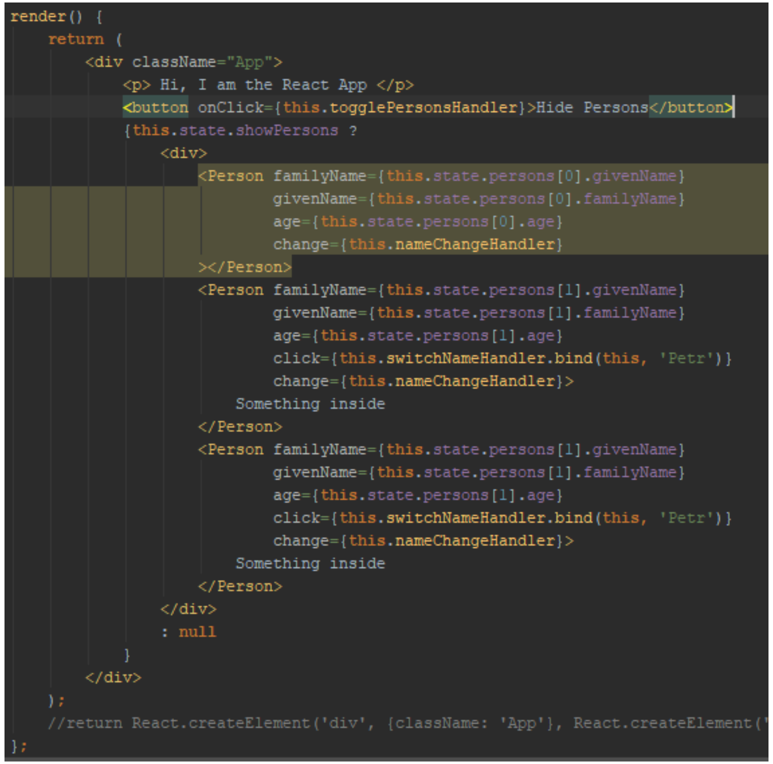
```
render() {
    return (
        <div className="App">
            <p> Hi, I am the React App </p>
            <button onClick={this.togglePersonsHandler}>Hide Persons</button>
            {this.state.showPersons ?
                <div>
                    <Person familyName={this.state.persons[0].givenName}
                            givenName={this.state.persons[0].familyName}
                            age={this.state.persons[0].age}
                            change={this.nameChangeHandler}
                    ></Person>
                    <Person familyName={this.state.persons[1].givenName}
                            givenName={this.state.persons[1].familyName}
                            age={this.state.persons[1].age}
                            click={this.switchNameHandler.bind(this, 'Petr')}
                            change={this.nameChangeHandler}>
                        Something inside
                    </Person>
                    <Person familyName={this.state.persons[1].givenName}
                            givenName={this.state.persons[1].familyName}
                            age={this.state.persons[1].age}
                            click={this.switchNameHandler.bind(this, 'Petr')}
                            change={this.nameChangeHandler}>
                        Something inside
                    </Person>
                </div>
                : null
            }
        </div>
    );
    //return React.createElement('div', {className: 'App'}, React.createElement(
};
```

Button Click To Show or Hide Persons Conditionally

- Since we are on the class-based component it merges only the showPersons property



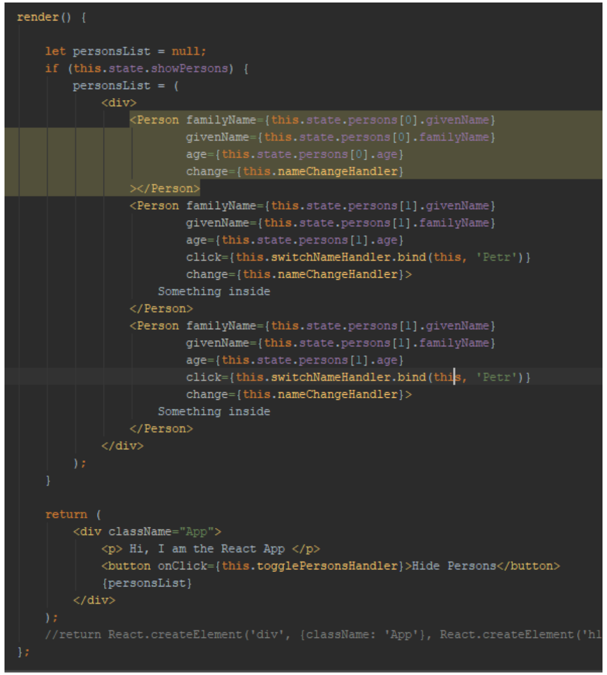
```
togglePersonsHandler = () => {
    const doesShow = this.state.showPersons;
    this.setState( state: {showPersons: !doesShow});
};
```



```
render() {
    return (
        <div className="App">
            <p> Hi, I am the React App </p>
            <button onClick={this.togglePersonsHandler}>Hide Persons</button>
            {this.state.showPersons ?
                <div>
                    <Person familyName={this.state.persons[0].givenName}
                            givenName={this.state.persons[0].familyName}
                            age={this.state.persons[0].age}
                            change={this.nameChangeHandler}
                    ></Person>
                    <Person familyName={this.state.persons[1].givenName}
                            givenName={this.state.persons[1].familyName}
                            age={this.state.persons[1].age}
                            click={this.switchNameHandler.bind(this, 'Petr')}
                            change={this.nameChangeHandler}>
                        Something inside
                    </Person>
                    <Person familyName={this.state.persons[1].givenName}
                            givenName={this.state.persons[1].familyName}
                            age={this.state.persons[1].age}
                            click={this.switchNameHandler.bind(this, 'Petr')}
                            change={this.nameChangeHandler}>
                        Something inside
                    </Person>
                </div>
                : null
            }
        </div>
    );
    //return React.createElement('div', {className: 'App'}, React.createElement('
};
```

IF Conditionals Another Alternative

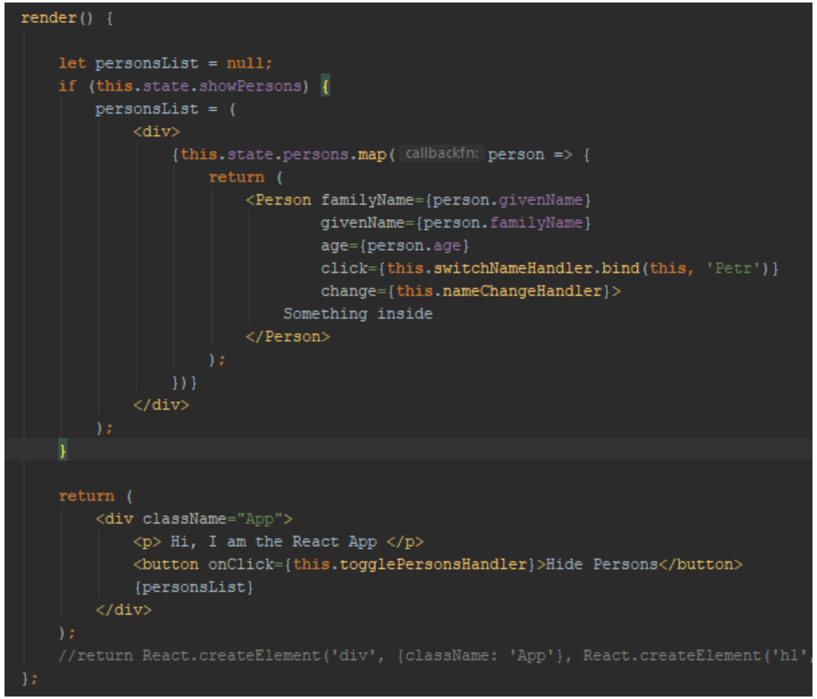
- We can also “outsource” the IF check in the render function but before the return element
  - This is the preferred variant of doing the conditionals in React



```
render() {

    let personsList = null;
    if (this.state.showPersons) {
        personsList = (
            <div>
                <Person familyName={this.state.persons[0].givenName}
                        givenName={this.state.persons[0].familyName}
                        age={this.state.persons[0].age}
                        change={this.nameChangeHandler}
                ></Person>
                <Person familyName={this.state.persons[1].givenName}
                        givenName={this.state.persons[1].familyName}
                        age={this.state.persons[1].age}
                        click={this.switchNameHandler.bind(this, 'Petr')}
                        change={this.nameChangeHandler}>
                    Something inside
                </Person>
                <Person familyName={this.state.persons[1].givenName}
                        givenName={this.state.persons[1].familyName}
                        age={this.state.persons[1].age}
                        click={this.switchNameHandler.bind(this, 'Petr')}
                        change={this.nameChangeHandler}>
                    Something inside
                </Person>
            </div>
        );
    }

    return (
        <div className="App">
            <p> Hi, I am the React App </p>
            <button onClick={this.togglePersonsHandler}>Hide Persons</button>
            {personsList}
        </div>
    );
    //return React.createElement('div', {className: 'App'}, React.createElement('h1
};
```

Return the List Objects From State Using Map

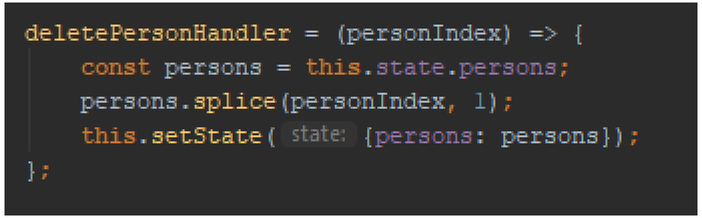
- Here, we in the case of click show the persons from the class-based component state



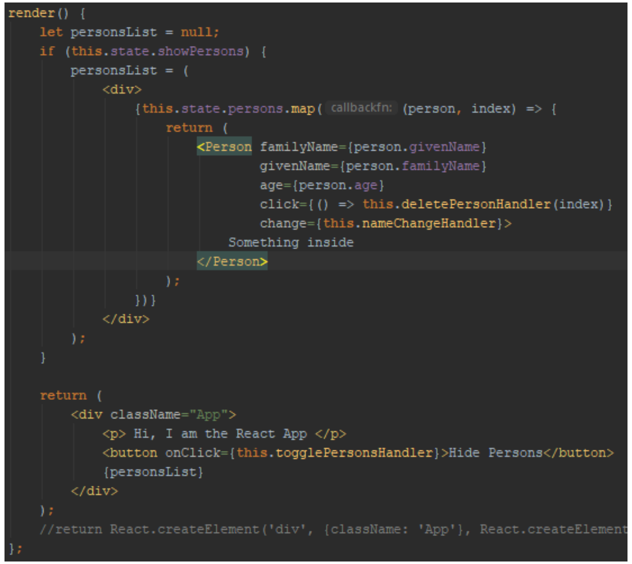
```
render() {

    let personsList = null;
    if (this.state.showPersons) {
        personsList = (
            <div>
                {this.state.persons.map( callbackfn: person => {
                    return (
                        <Person familyName={person.givenName}
                                givenName={person.familyName}
                                age={person.age}
                                click={this.switchNameHandler.bind(this, 'Petr')}
                                change={this.nameChangeHandler}>
                            Something inside
                        </Person>
                    );
                })}
            </div>
        );
    }

    return (
        <div className="App">
            <p> Hi, I am the React App </p>
            <button onClick={this.togglePersonsHandler}>Hide Persons</button>
            {personsList}
        </div>
    );
    //return React.createElement('div', {className: 'App'}, React.createElement('h1'
};
```

Manipulating Lists in the Component State

- This is not the ideal way of how to remove a person from a React State
    - But it works…
    - The **splice** method removes elements from an array
- The drawback of this method is that when I do this I am manipulating original state (in persons.splice) and if an error occurs it will lead to the unpredictable apps



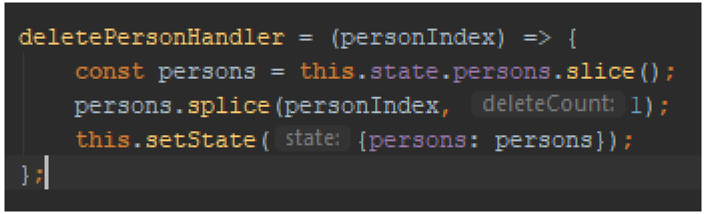
```
deletePersonHandler = (personIndex) => {
    const persons = this.state.persons;
    persons.splice(personIndex, 1);
    this.setState( state: {persons: persons});
};
```



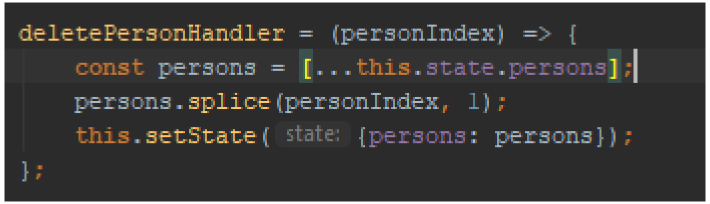
```
render() {
    let personsList = null;
    if (this.state.showPersons) {
        personsList = (
            <div>
                {this.state.persons.map( callbackfn: (person, index) => {
                    return (
                        <Person familyName={person.givenName}
                                givenName={person.familyName}
                                age={person.age}
                                click={() => this.deletePersonHandler(index)}
                                change={this.nameChangeHandler}>
                            Something inside
                        </Person>
                    );
                })}
            </div>
        );
    }

    return (
        <div className="App">
            <p> Hi, I am the React App </p>
            <button onClick={this.togglePersonsHandler}>Hide Persons</button>
            {personsList}
        </div>
    );
    //return React.createElement('div', {className: 'App'}, React.createElement
};
```

Manipulating Lists in the Component State - Correct Way

- The better way is to create a copy of that array
- There are several ways of how to do it
  - First, call a **slice()** method as is shown in the picture on the right top of this slide
  - Using ES6 feature **[...this.state.persons]**
- The state should always be updated without mutating the original state



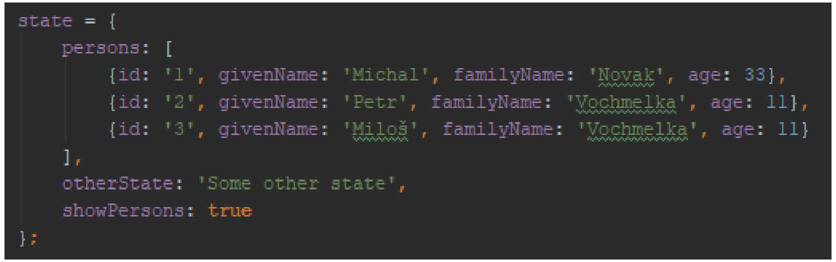
```
deletePersonHandler = (personIndex) => {
    const persons = this.state.persons.slice();
    persons.splice(personIndex,  deleteCount: 1);
    this.setState( state: {persons: persons});
};
```



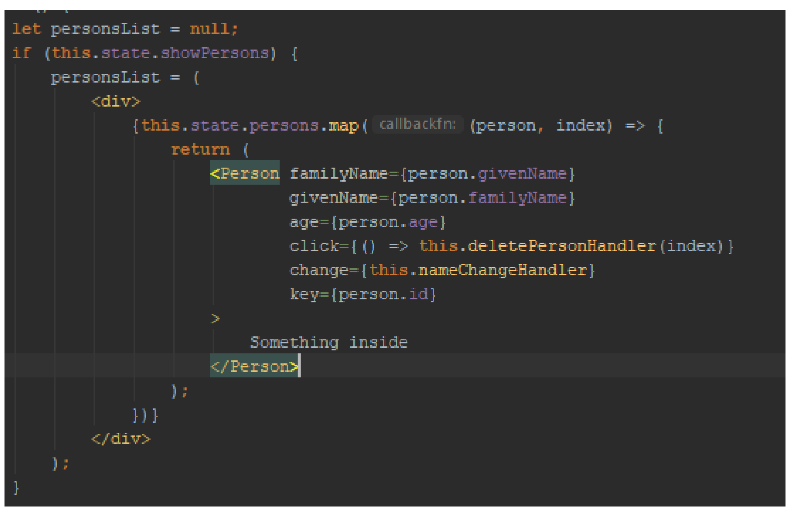
```
deletePersonHandler = (personIndex) => {
    const persons = [...this.state.persons];
    persons.splice(personIndex, 1);
    this.setState( state: {persons: persons});
};
```

How the Lists Are re-rendered?

- Without any keys the React will re-render the whole lists that is not a lot efficient
- To allow to keep track of individual elements we can add a key to only re-render the elements that changed
    - This can be done using the ID from the database
    - In our example, we add **key={person.id}**



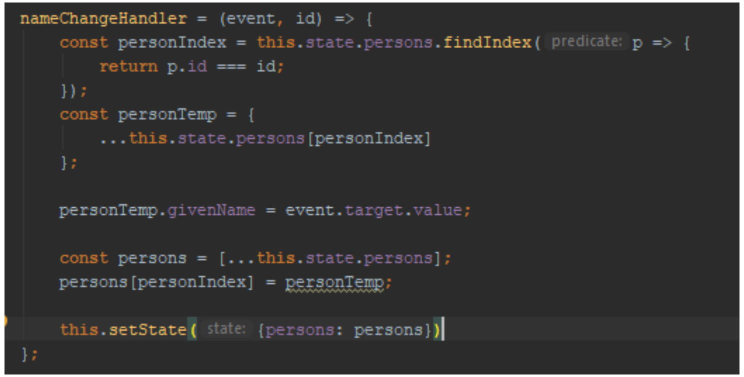
```
state = {
    persons: [
        {id: '1', givenName: 'Michal', familyName: 'Novak', age: 33},
        {id: '2', givenName: 'Petr', familyName: 'Vochmelka', age: 11},
        {id: '3', givenName: 'Miloš', familyName: 'Vochmelka', age: 11}
    ],
    otherState: 'Some other state',
    showPersons: true
};
```



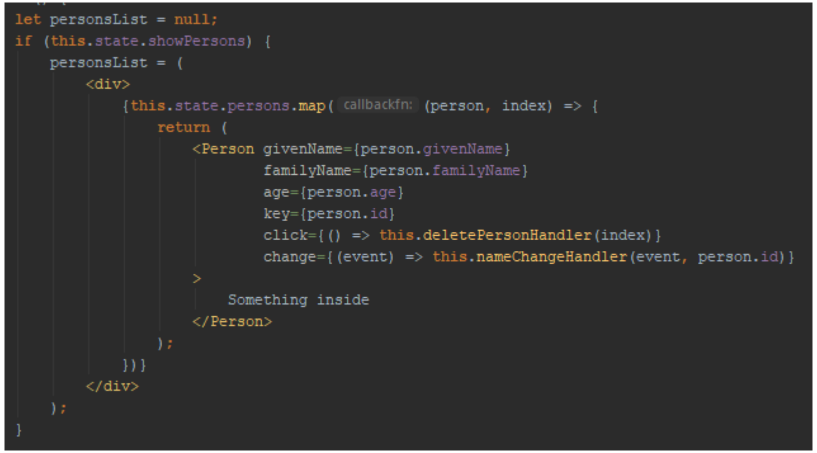
```
let personsList = null;
if (this.state.showPersons) {
    personsList = (
        <div>
            {this.state.persons.map( callbackfn: (person, index) => {
                return (
                    <Person familyName={person.givenName}
                            givenName={person.familyName}
                            age={person.age}
                            click={() => this.deletePersonHandler(index)}
                            change={this.nameChangeHandler}
                            key={person.id}
                    >
                        Something inside
                    </Person>
                );
            })}
        </div>
    );
}
```

How the Lists Are re-rendered?

- Set event for name handler to a specific person (based on the person.id)
- As you can see, always work on the copies of the object states



```
nameChangeHandler = (event, id) => {
    const personIndex = this.state.persons.findIndex( predicate: p => {
        return p.id === id;
    });
    const personTemp = {
        ...this.state.persons[personIndex]
    };

    personTemp.givenName = event.target.value;

    const persons = [...this.state.persons];
    persons[personIndex] = personTemp;

    this.setState( state: {persons: persons})
};
```



```
let personsList = null;
if (this.state.showPersons) {
    personsList = (
        <div>
            {this.state.persons.map( callbackfn: (person, index) => {
                return (
                    <Person givenName={person.givenName}
                            familyName={person.familyName}
                            age={person.age}
                            key={person.id}
                            click={() => this.deletePersonHandler(index)}
                            change={(event) => this.nameChangeHandler(event, person.id)}
                    >
                        Something inside
                    </Person>
                );
            })}
        </div>
    );
}
```

Tasks

1. Create an input field (in App component) with a change listener which outputs the length of the entered text below it (e.g., in a paragraph)
2. Create a new component (=> ValidationComponent) which receives the text length as a prop
3. Inside the ValidationComponent, either output “Text too short” or “Text too long enough” depending on the text length (e.g., take 10 as a minimum length)

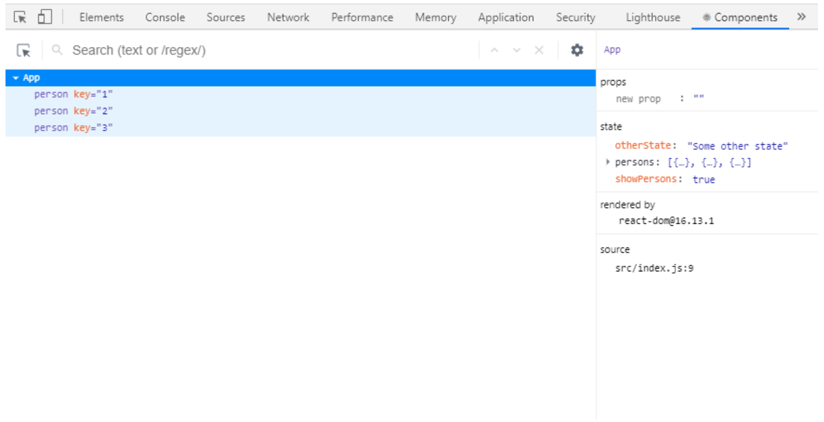
Section: Debugging

Developer Console

- Error messages can be debugged using Developer Console
  - In **Mozilla Firefox** and **Google Chrome** the shortcut is “**ctrl+shift+i**”
- Handling logical errors
  - In Google Chrome Developer Console find Sources tab
    - In this tab we can find a local copy of a source code
    - We can add a breakpoint here and debug it as we know it from the IDEs
- Debug code using **console.log(something)**

React Developer Tools

- Install add-on ‘React Developer Tools’ (Google Chrome/Firefox)
- In Google Chrome, it can looks as follows:
    - It shows the props that it receives, state that it manages, etc.
    - We can edit these fields etc.



```
▾ App
    person key="1"
    person key="2"
    person key="3"
```

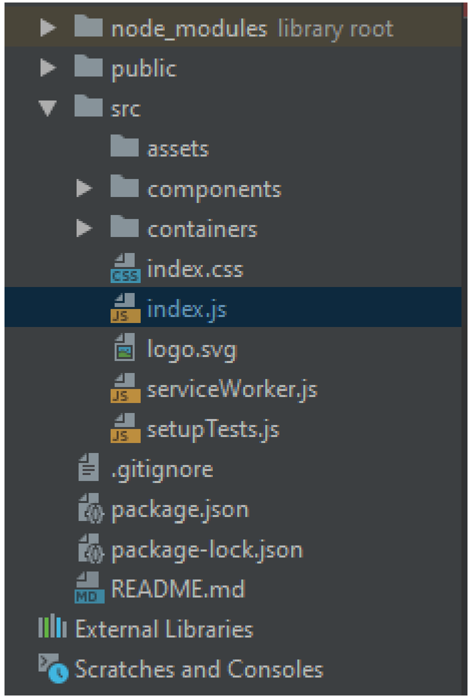
```
App

props
  new prop   : ""

state
  otherState: "Some other state"
▸ persons: [{…}, {…}, {…}]
  showPersons: true

rendered by
  react-dom@16.13.1

source
  src/index.js:9
```

Tasks

1. Install React Developer Tools
2. Play with the debugging options (place it whenever possible and check what is in the web-console)

Section: Project Structure & React Internals

Basic Project Structure


```
▶ node_modules library root
▶ public
▼ src
      assets
    ▶ components
    ▶ containers
      index.css
      index.js
      logo.svg
      serviceWorker.js
      setupTests.js
  .gitignore
  package.json
  package-lock.json
  README.md
External Libraries
Scratches and Consoles
```

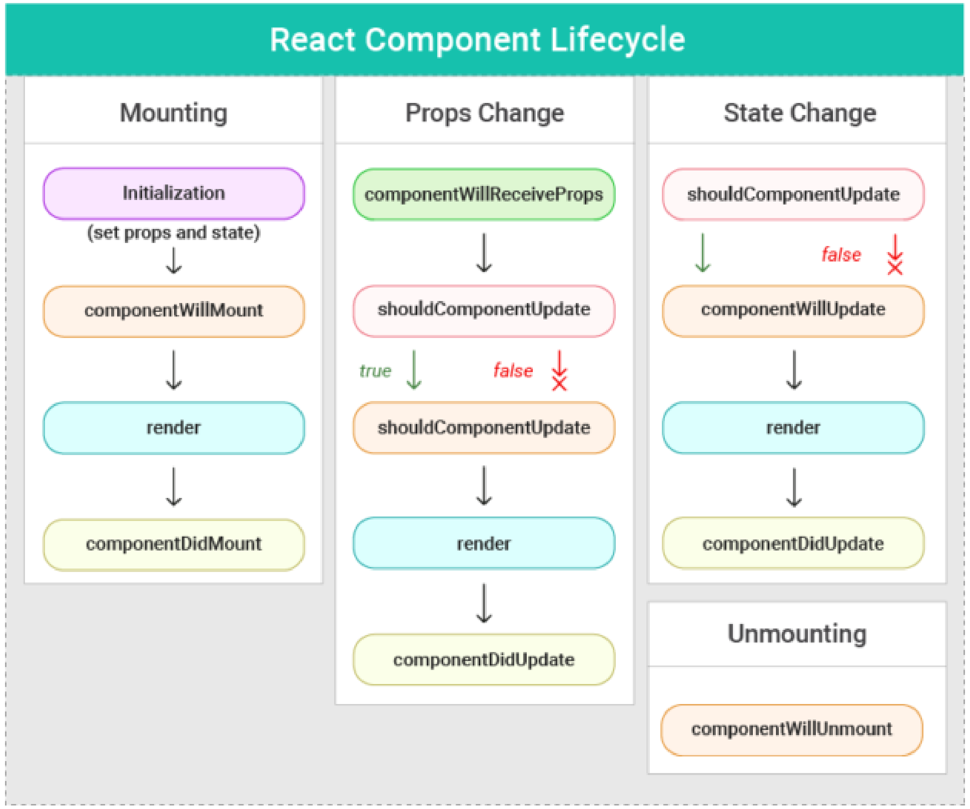
- Basically, we can divide our application into:
  - Assets (pictures etc.)
  - Components (persons, …)
  - Containers (that holds the components)
- It is a good practice to create such a structure to easily navigate through the project components in a large projects
- Further, we will add additional folders (context, services, domain, …)

Class-based vs Functional Components

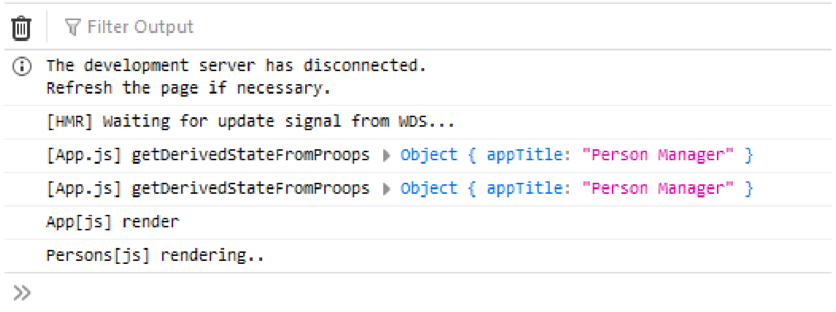
class-based	Functional
class XY extends Component	const XY = (props) { … }
Access to Store	Access to state (useState())
Lifecycle Hooks	Access Props via “props”
Access State and Props via “this”	props.XY
this.state.XY & this.props.XY	

Component Lifecycle

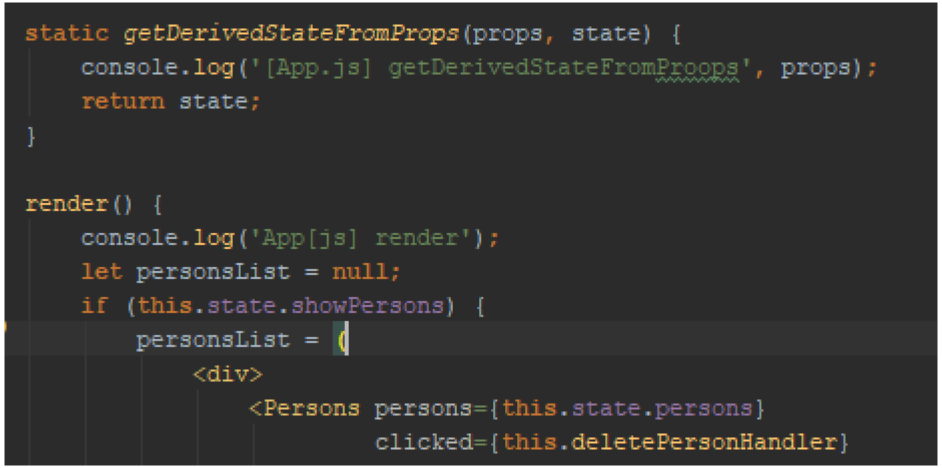
- These methods are executed for us by React
- These are used to fetch data from the web or for some clean-up work



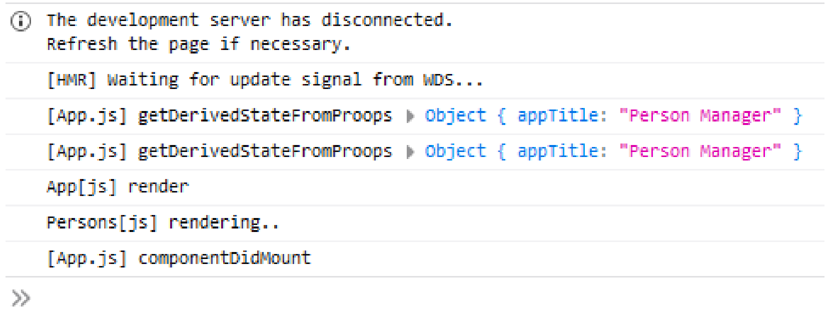
Component Creation Lifecycle in Action



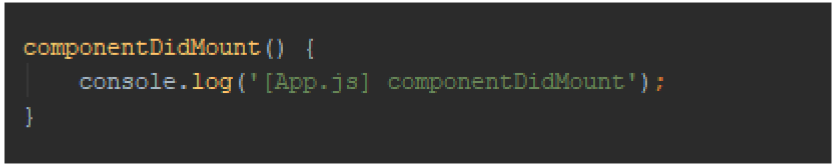
```
Filter Output
The development server has disconnected.
Refresh the page if necessary.
[HMR] Waiting for update signal from WDS...
[App.js] getDerivedStateFromProops ▸ Object { appTitle: "Person Manager" }
[App.js] getDerivedStateFromProops ▸ Object { appTitle: "Person Manager" }
App[js] render
Persons[js] rendering..
»
```



```
static getDerivedStateFromProps(props, state) {
    console.log('[App.js] getDerivedStateFromProops', props);
    return state;
}

render() {
    console.log('App[js] render');
    let personsList = null;
    if (this.state.showPersons) {
        personsList = (
            <div>
                <Persons persons={this.state.persons}
                         clicked={this.deletePersonHandler}
```

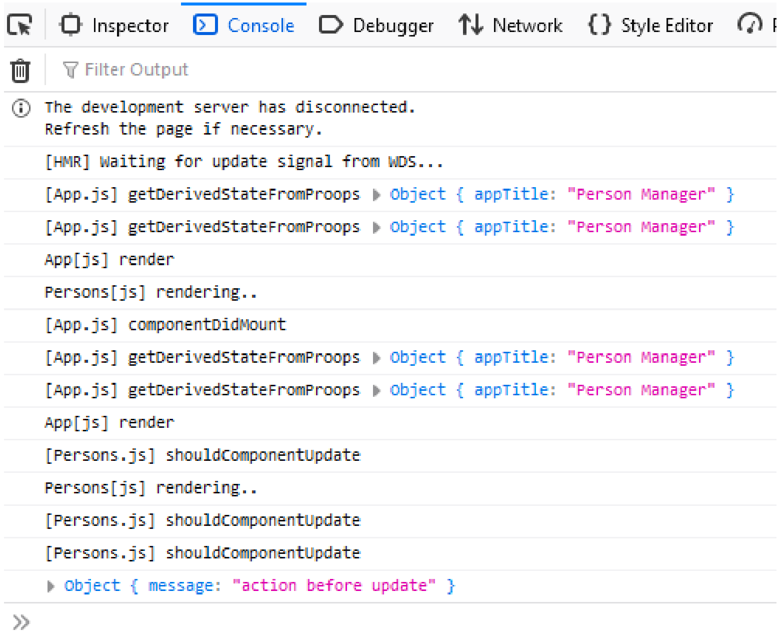


```
The development server has disconnected.
Refresh the page if necessary.
[HMR] Waiting for update signal from WDS...
[App.js] getDerivedStateFromProops ▸ Object { appTitle: "Person Manager" }
[App.js] getDerivedStateFromProops ▸ Object { appTitle: "Person Manager" }
App[js] render
Persons[js] rendering..
[App.js] componentDidMount
»
```

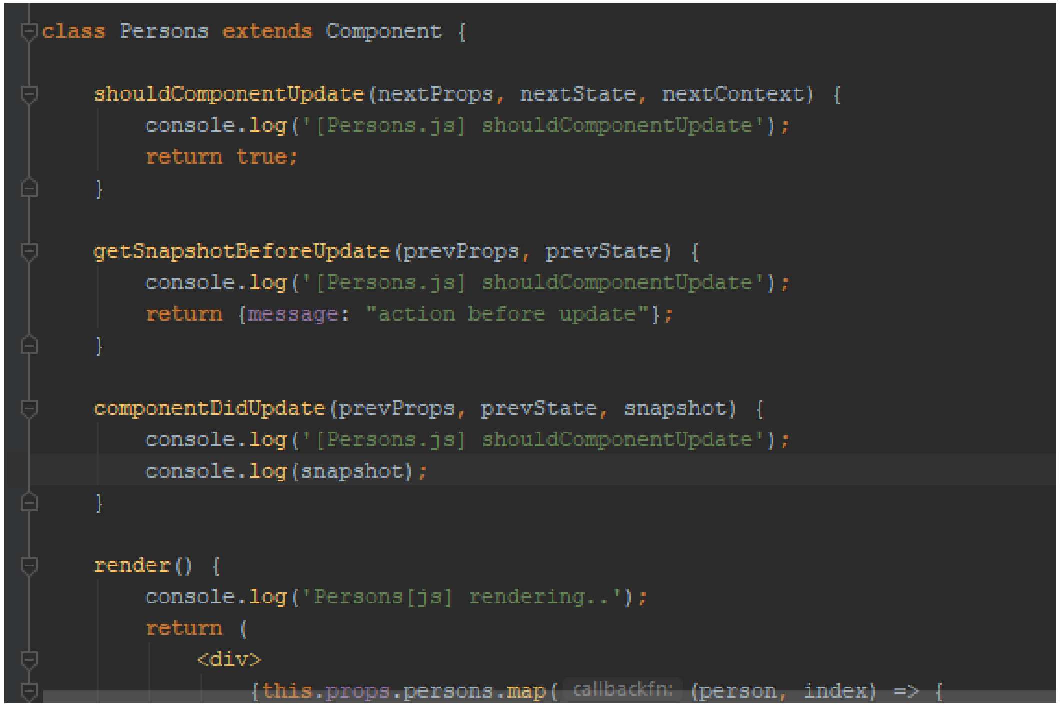


```
componentDidMount() {
    console.log('[App.js] componentDidMount');
}
```

Component Update Lifecycle (for props changes)



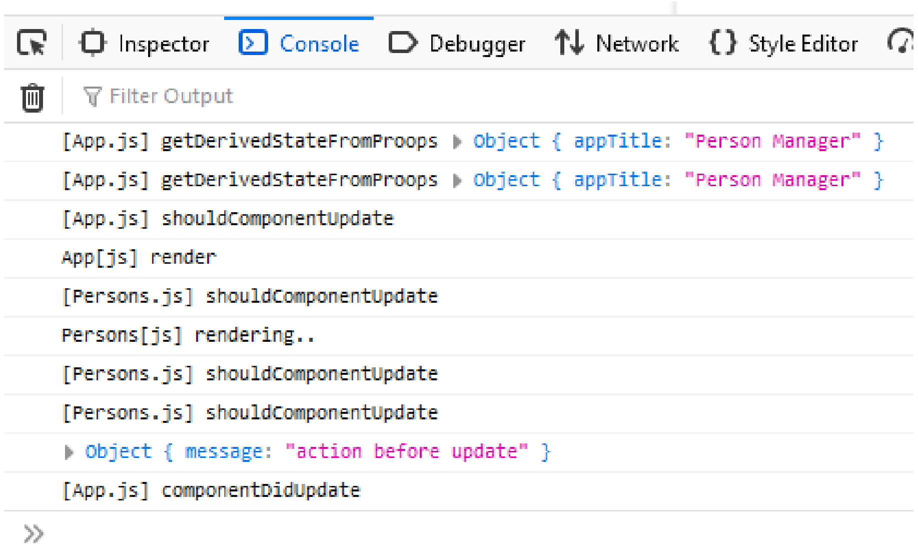
```
The development server has disconnected.
Refresh the page if necessary.
[HMR] Waiting for update signal from WDS...
[App.js] getDerivedStateFromPoops ▶ Object { appTitle: "Person Manager" }
[App.js] getDerivedStateFromPoops ▶ Object { appTitle: "Person Manager" }
App[js] render
Persons[js] rendering..
[App.js] componentDidMount
[App.js] getDerivedStateFromPoops ▶ Object { appTitle: "Person Manager" }
[App.js] getDerivedStateFromPoops ▶ Object { appTitle: "Person Manager" }
App[js] render
[Persons.js] shouldComponentUpdate
Persons[js] rendering..
[Persons.js] shouldComponentUpdate
[Persons.js] shouldComponentUpdate
▶ Object { message: "action before update" }
```



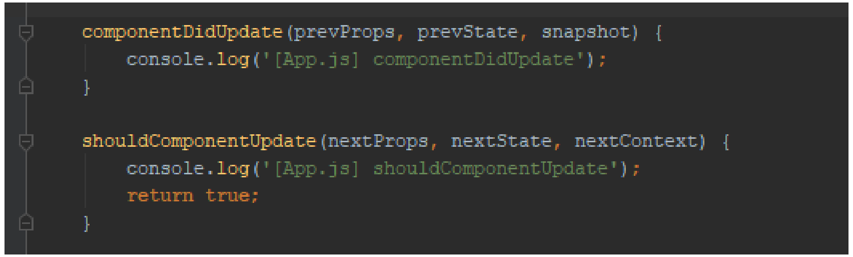
```
class Persons extends Component {

    shouldComponentUpdate(nextProps, nextState, nextContext) {
        console.log('[Persons.js] shouldComponentUpdate');
        return true;
    }

    getSnapshotBeforeUpdate(prevProps, prevState) {
        console.log('[Persons.js] shouldComponentUpdate');
        return {message: "action before update"};
    }

    componentDidUpdate(prevProps, prevState, snapshot) {
        console.log('[Persons.js] shouldComponentUpdate');
        console.log(snapshot);
    }

    render() {
        console.log('Persons[js] rendering..');
        return (
            <div>
                {this.props.persons.map( callbackfn: (person, index) => {
```

Component Update Lifecycle (for state changes)

- In App class



```
[App.js] getDerivedStateFromProops ▶ Object { appTitle: "Person Manager" }
[App.js] getDerivedStateFromProops ▶ Object { appTitle: "Person Manager" }
[App.js] shouldComponentUpdate
App[js] render
[Persons.js] shouldComponentUpdate
Persons[js] rendering..
[Persons.js] shouldComponentUpdate
[Persons.js] shouldComponentUpdate
▶ Object { message: "action before update" }
[App.js] componentDidUpdate
```

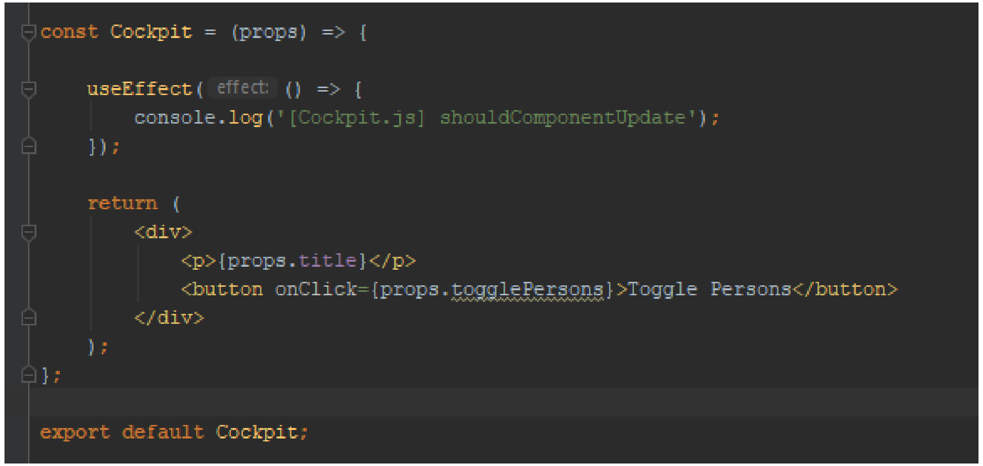


```
componentDidUpdate(prevProps, prevState, snapshot) {
    console.log('[App.js] componentDidUpdate');
}

shouldComponentUpdate(nextProps, nextState, nextContext) {
    console.log('[App.js] shouldComponentUpdate');
    return true;
}
```

- I can see that after updating all the elements the App component is also updated
- To review the methods that represents the Component Lifecycle visit the following page:
- https://engineering.musefind.com/react-lifecycle-methods-how-and-when-to-use-them-2111a1b692b1

Component useEffect

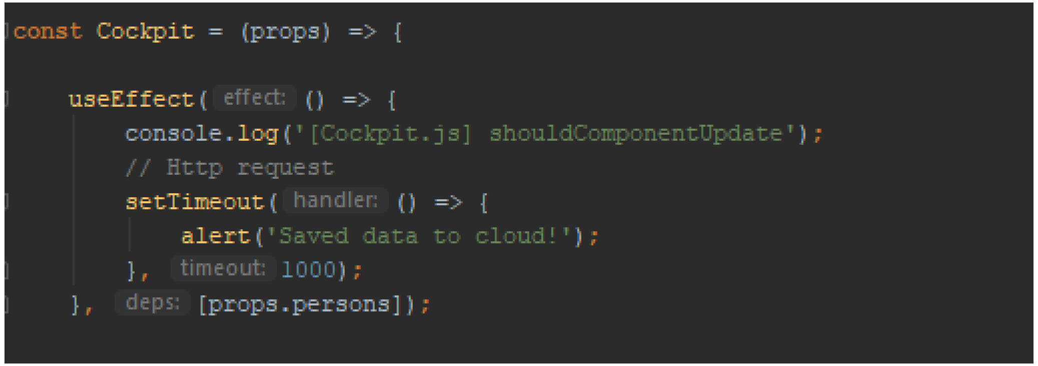
- useEffect() function is called every time the component is re-rendered in the Virtual Dom
- For example to sent a HTTP Request when the component is re-rendered



```
const Cockpit = (props) => {

    useEffect( effect: () => {
        console.log('[Cockpit.js] shouldComponentUpdate');
    });

    return (
        <div>
            <p>{props.title}</p>
            <button onClick={props.togglePersons}>Toggle Persons</button>
        </div>
    );
};

export default Cockpit;
```

Component useEffect - How control when its executed

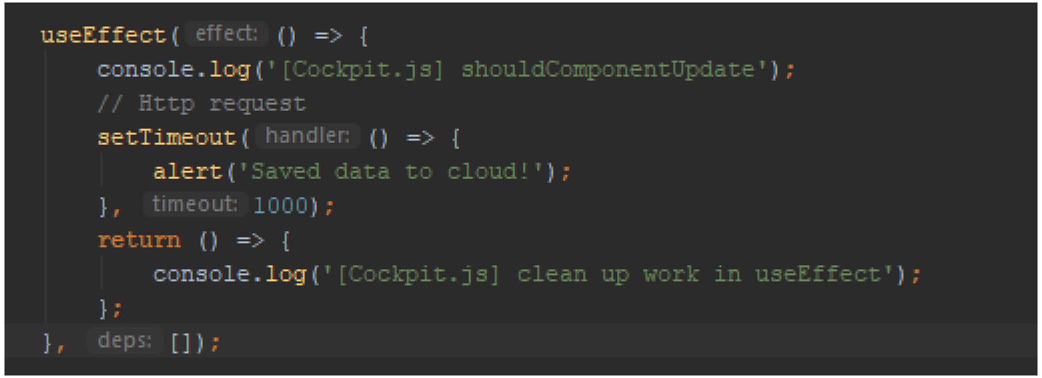
- We can add additional parameter that sets that this component **is re-rendered only when some props changes**
- **If** this parameter is **empty array []**, this tells React that this effect has no dependencies -> It re-renders **only on the first render** of this Component



```
const Cockpit = (props) => {

    useEffect( effect: () => {
        console.log('[Cockpit.js] shouldComponentUpdate');
        // Http request
        setTimeout( handler: () => {
            alert('Saved data to cloud!');
        }, timeout: 1000);
    }, deps: [props.persons]);
```

Component useEffect - Clean up Work

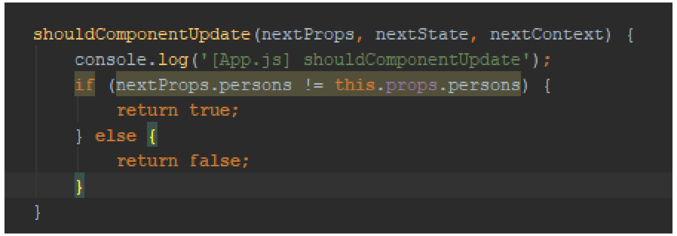
- We can use return as additional argument to do a clean up work in useEffect



```
useEffect( effect: () => {
    console.log('[Cockpit.js] shouldComponentUpdate');
    // Http request
    setTimeout( handler: () => {
        alert('Saved data to cloud!');
    }, timeout: 1000);
    return () => {
        console.log('[Cockpit.js] clean up work in useEffect');
    };
}, deps: []);
```

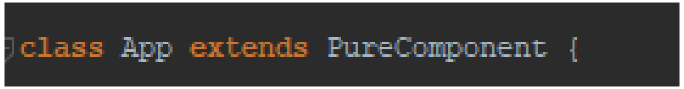
Optimizing re-rendering (class-based components)

- In **shouldComponentUpdate**, we can check what properties did changed
  - Pay special attention here, that we are comparing the pointers (in our use, case of persons)
  - If we do not use:
    - const persons = [...this.state.persons], the reference would be the same and this check would not
- If you want to check all the props in the class, consider extending **PureComponent** (if you do this, you do not need to implement shouldComponentUpdate, and it will check for all the props if there is any change



```
shouldComponentUpdate(nextProps, nextState, nextContext) {
    console.log('[App.js] shouldComponentUpdate');
    if (nextProps.persons != this.props.persons) {
        return true;
    } else {
        return false;
    }
}
```

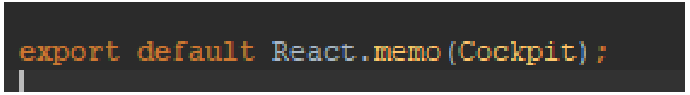
- Check what re-renders in Chrome:
  - -> Developer Console -> More tools -> Rendering -> Paint flashing (enable)



```
class App extends PureComponent {
```

Optimizing re-rendering (functional components)

- You can wrap your function in export part to `React.memo(xyz)`



```
export default React.memo(Cockpit);
```

- When some props change in that functional components it re-renders
- For example, if I write some text in input form in parent (e.g., class-based component) that do not touch that functional component, that action would not trigger any change on the child functional component

When should you optimize?

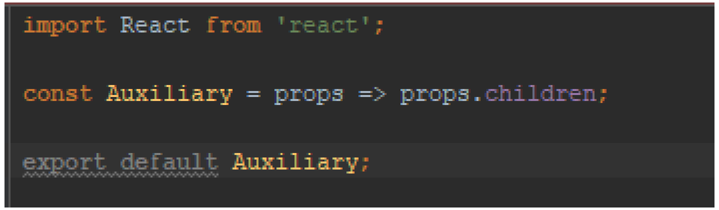
- From the previous slides, it may seem that I should add **React.memo(xyz)** or **shouldComponentUpdate(...)** to all of the classes
- But actually, it isn’t
  - It takes something to compute all the checks in **shouldComponentUpdate, …**
  - You should evaluate carefully where it makes sense to add these checks to re-rendering the application

How React Updates the DOM?

- The render() method is just a suggestion to render the component (for Virtual Dom)
  - shouldComponentUpdate() -> passed
- It compares Old Virtual DOM with Re-rendered Virtual DOM (created when render() method is called) -> faster than “real” DOM
  - It detects differences -> If differences found => update real DOM
  - render() method does not immediately update the real DOM
  - Accessing the real DOM is really slow

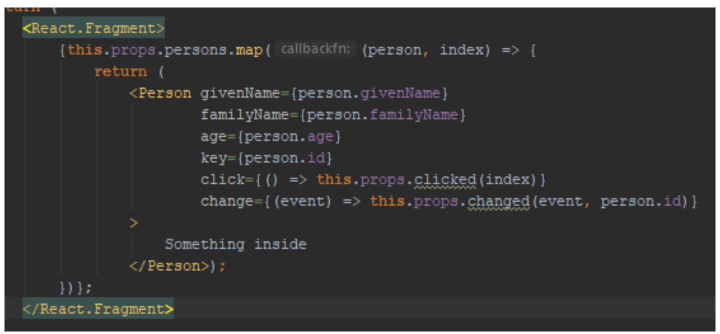
How to Retrieve More <div> or Other Elements ?

- You can create your own wrapping element (High Order Components -- HOC)
- Create HOC component such like this:

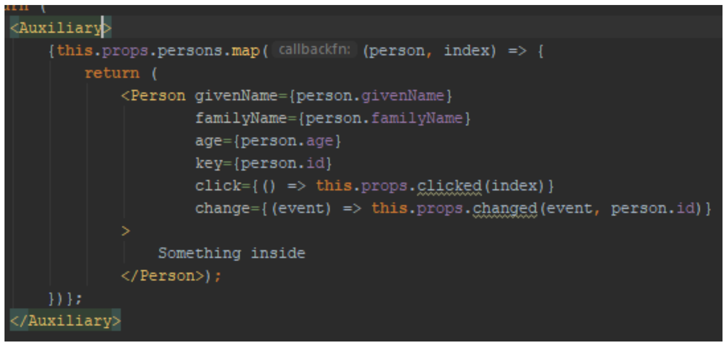


```
import React from 'react';

const Auxiliary = props => props.children;

export default Auxiliary;
```

- props.children refers to all the inner elements
- Since 16.2 there is a built-in React HOC component (React.Fragment)



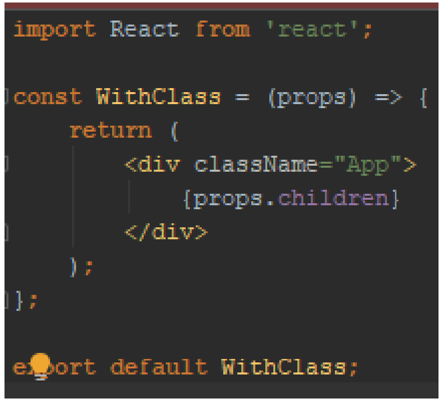
```
<React.Fragment>
    {this.props.persons.map( callbackfn: (person, index) => {
        return (
            <Person givenName={person.givenName}
                    familyName={person.familyName}
                    age={person.age}
                    key={person.id}
                    click={() => this.props.clicked(index)}
                    change={(event) => this.props.changed(event, person.id)}
            >
                Something inside
            </Person>);
    })};
</React.Fragment>
```



```
<Auxiliary>
    {this.props.persons.map( callbackfn: (person, index) => {
        return (
            <Person givenName={person.givenName}
                    familyName={person.familyName}
                    age={person.age}
                    key={person.id}
                    click={() => this.props.clicked(index)}
                    change={(event) => this.props.changed(event, person.id)}
            >
                Something inside
            </Person>);
    })};
</Auxiliary>
```

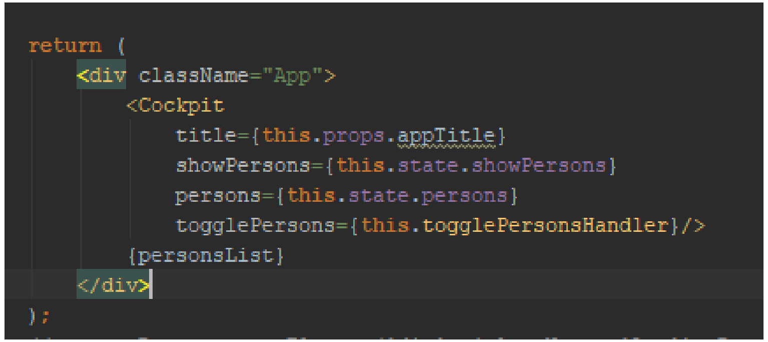
High Order Components

- It just wraps another component
- Using HOC setting, e.g., some styling

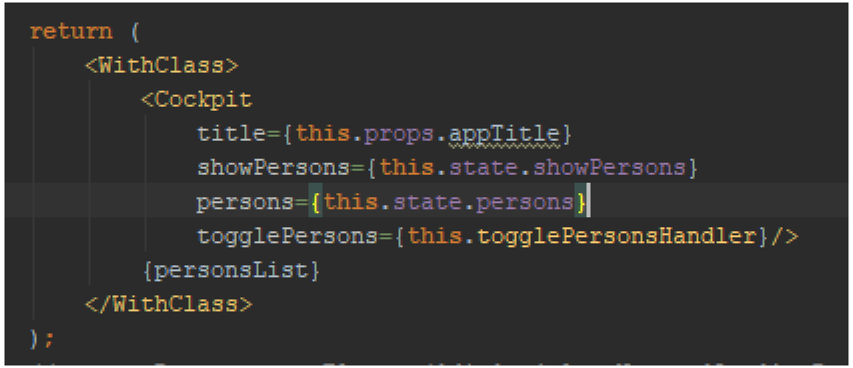


```
import React from 'react';

const WithClass = (props) => {
    return (
        <div className="App">
            {props.children}
        </div>
    );
};

export default WithClass;
```

Original class



```
return (
    <div className="App">
        <Cockpit
            title={this.props.appTitle}
            showPersons={this.state.showPersons}
            persons={this.state.persons}
            togglePersons={this.togglePersonsHandler}/>
        {personsList}
    </div>
);
```

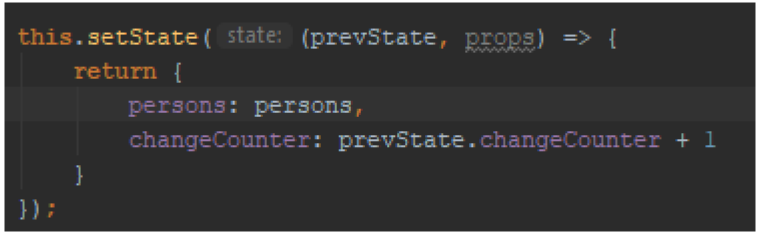


```
return (
    <WithClass>
        <Cockpit
            title={this.props.appTitle}
            showPersons={this.state.showPersons}
            persons={this.state.persons}
            togglePersons={this.togglePersonsHandler}/>
        {personsList}
    </WithClass>
);
```

Using HOC

Setting State Correctly

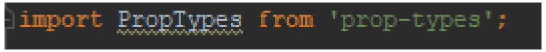
- Consider the situation that you want to modify some state in **setState** method
  - For example, you will set counter to count entered keystrokes to input field
  - Then you can call setState with the following parameters.. This will avoid possible mistakes
  - It is not appropriate to use any kind of **this.state..** In this.setState method, since it is not called synchronously and it may lead to the unexpected mistakes



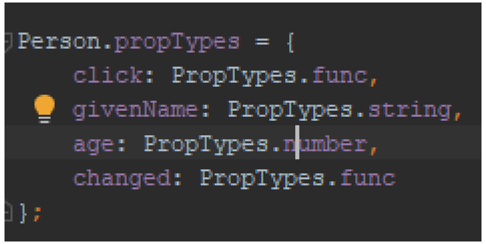
```
this.setState( state: (prevState, props) => {
    return {
        persons: persons,
        changeCounter: prevState.changeCounter + 1
    }
});
```

Checking Input Props (class-based and func. comp.)

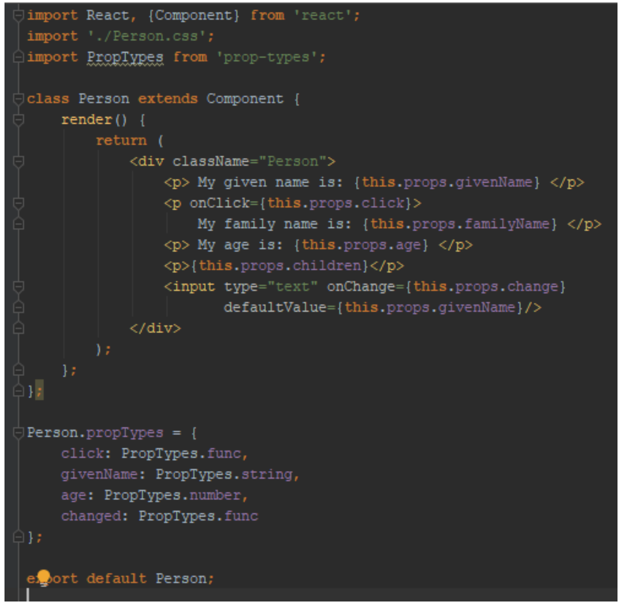
- Throw an error if someone pass incorrect props to your component (useful in the case if you are creating a library or you are working in a large team)
  - Run in terminal> `npm install --save prop-types`
  - It will check if you pass correct props



```
import PropTypes from 'prop-types';
```



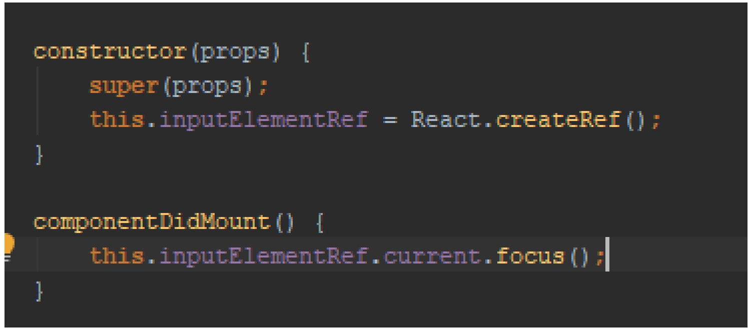
```
Person.propTypes = {
    click: PropTypes.func,
    givenName: PropTypes.string,
    age: PropTypes.number,
    changed: PropTypes.func
};
```



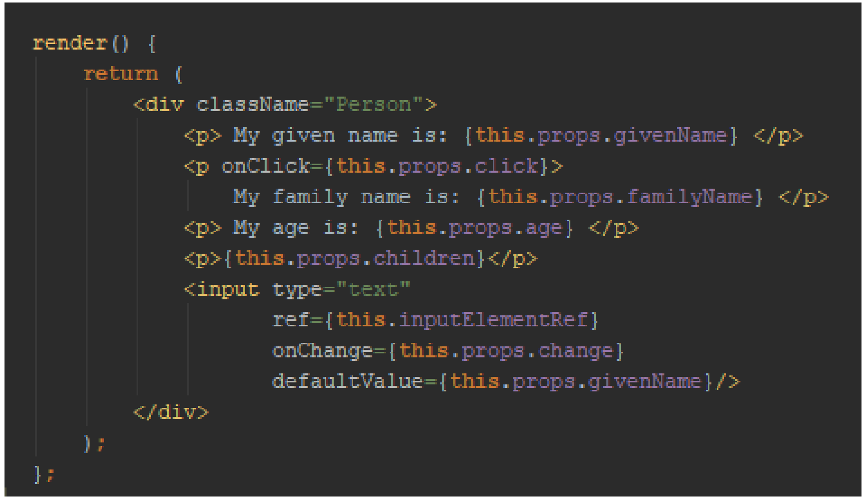
```
import React, {Component} from 'react';
import './Person.css';
import PropTypes from 'prop-types';

class Person extends Component {
    render() {
        return (
            <div className="Person">
                <p> My given name is: {this.props.givenName} </p>
                <p onClick={this.props.click}>
                    My family name is: {this.props.familyName} </p>
                <p> My age is: {this.props.age} </p>
                <p>{this.props.children}</p>
                <input type="text" onChange={this.props.change}
                       defaultValue={this.props.givenName}/>
            </div>
        );
    };
};

Person.propTypes = {
    click: PropTypes.func,
    givenName: PropTypes.string,
    age: PropTypes.number,
    changed: PropTypes.func
};

export default Person;
```

Refs in Class-based Components

- To, e.g., focus a specific element after reload use **ref**



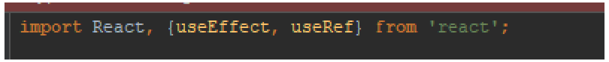
```
constructor(props) {
    super(props);
    this.inputElementRef = React.createRef();
}

componentDidMount() {
    this.inputElementRef.current.focus();
}
```



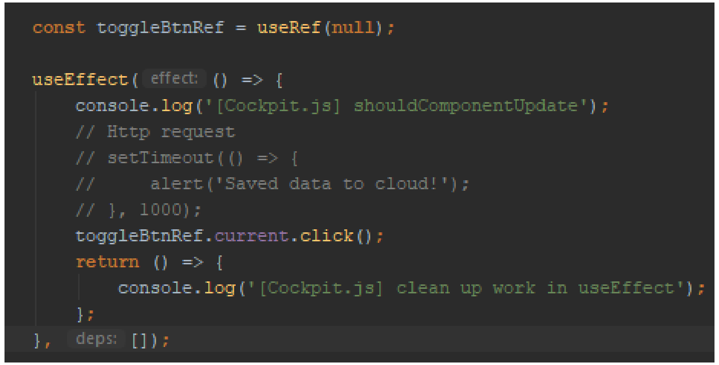
```
render() {
    return (
        <div className="Person">
            <p> My given name is: {this.props.givenName} </p>
            <p onClick={this.props.click}>
                My family name is: {this.props.familyName} </p>
            <p> My age is: {this.props.age} </p>
            <p>{this.props.children}</p>
            <input type="text"
                   ref={this.inputElementRef}
                   onChange={this.props.change}
                   defaultValue={this.props.givenName}/>
        </div>
    );
};
```

Refs in Functional Components

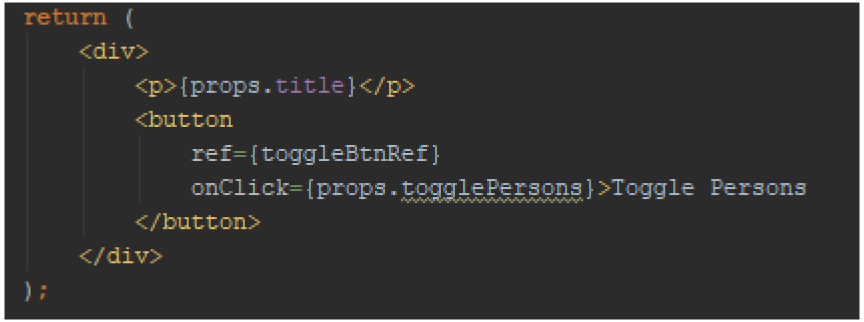
- To, e.g., click on a button after reload use



```
import React, {useEffect, useRef} from 'react';
```



```
const toggleBtnRef = useRef(null);

useEffect( effect: () => {
    console.log('[Cockpit.js] shouldComponentUpdate');
    // Http request
    // setTimeout(() => {
    //     alert('Saved data to cloud!');
    // }, 1000);
    toggleBtnRef.current.click();
    return () => {
        console.log('[Cockpit.js] clean up work in useEffect');
    };
},  deps: []);
```



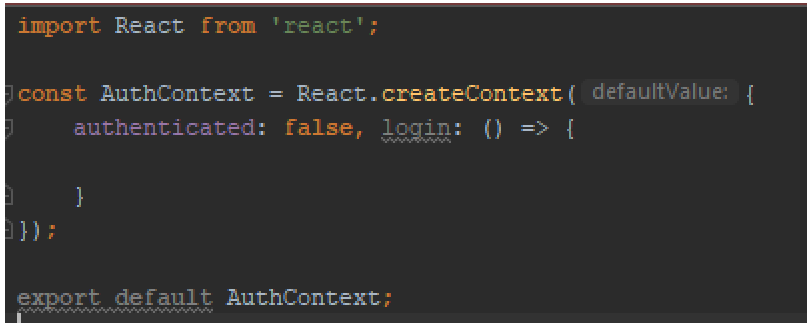
```
return (
    <div>
        <p>{props.title}</p>
        <button
            ref={toggleBtnRef}
            onClick={props.togglePersons}>Toggle Persons
        </button>
    </div>
);
```

Props Chain -- Problem

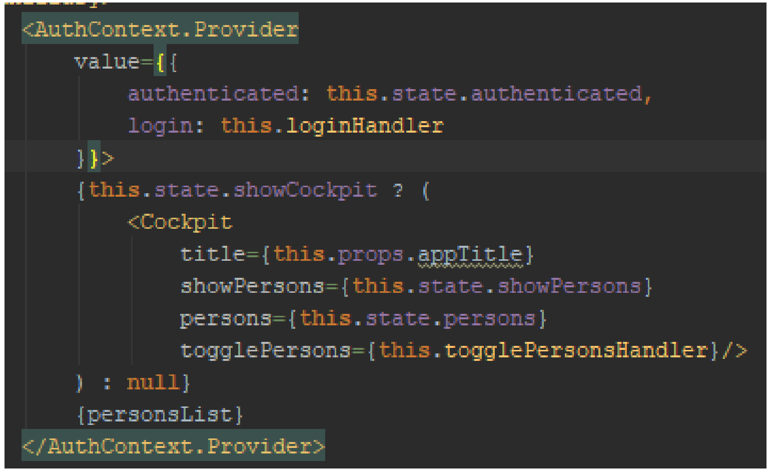
- Consider the situation where we need to pass information about authentication through several components
  - From App -> Persons -> Person
  - The Persons component just redirects the settings
  - In that case, we would have a lot of props that we just redirect between components and doing nothing with it
- The solution? => Context!

React Context API

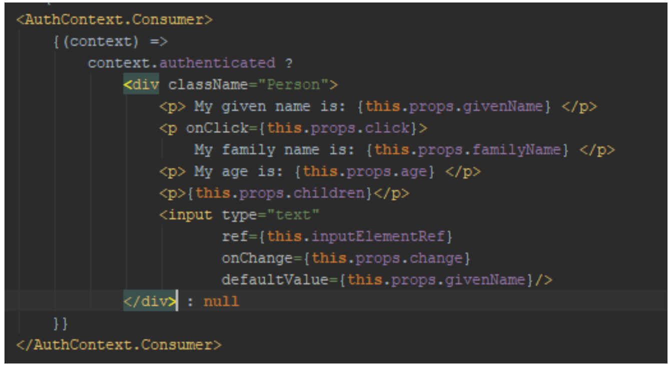
- React.createContext({})
  - It can be object, array, string, a number, …
- This component must wrap all the components that wants to use this React Context
  - As <AuthContext.Provider>
  - Or <AuthContext.Consumer> (+ func)



```
import React from 'react';

const AuthContext = React.createContext( defaultValue: {
    authenticated: false, login: () => {

    }
});

export default AuthContext;
```



```
<AuthContext.Provider
    value={{
        authenticated: this.state.authenticated,
        login: this.loginHandler
    }}>
    {this.state.showCockpit ? (
        <Cockpit
            title={this.props.appTitle}
            showPersons={this.state.showPersons}
            persons={this.state.persons}
            togglePersons={this.togglePersonsHandler}/>
    ) : null}
    {personsList}
</AuthContext.Provider>
```

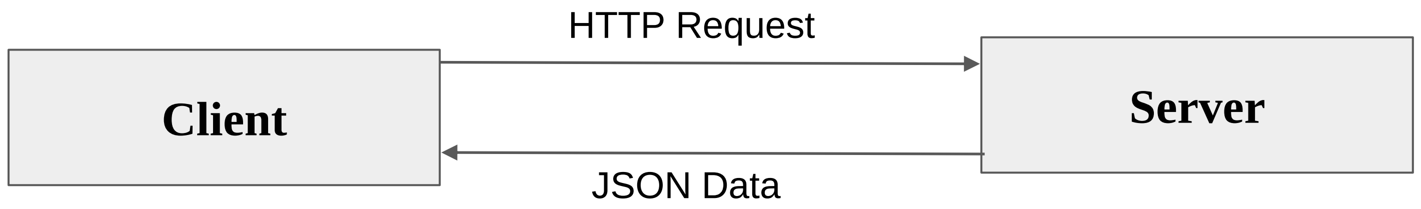


```
<AuthContext.Consumer>
    {(context) =>
        context.authenticated ?
            <div className="Person">
                <p> My given name is: {this.props.givenName} </p>
                <p onClick={this.props.click}>
                    My family name is: {this.props.familyName} </p>
                <p> My age is: {this.props.age} </p>
                <p>{this.props.children}</p>
                <input type="text"
                       ref={this.inputElementRef}
                       onChange={this.props.change}
                       defaultValue={this.props.givenName}/>
            </div> : null
    }}
</AuthContext.Consumer>
```

Tasks

1. Implement methods as componentDidMount() and others that are mentioned in the slides previously
2. Play with that methods (output console.log whenever possible), check the order of the lifecycle methods invocation
3. Create your own Auxiliary component to wrap several <div> methods in predefined class-based component
4. Install and use PropTypes to check input parameters in arbitrarily component
5. Optimize the code so that you will re-render when all the state or props changed
   a. Optimize the code so that you will re-render the component only if the predefined field changed
   b. Use all the presented approaches and check the re-rendered components using the presented plugin

Section: HTTP Requests / AJAX

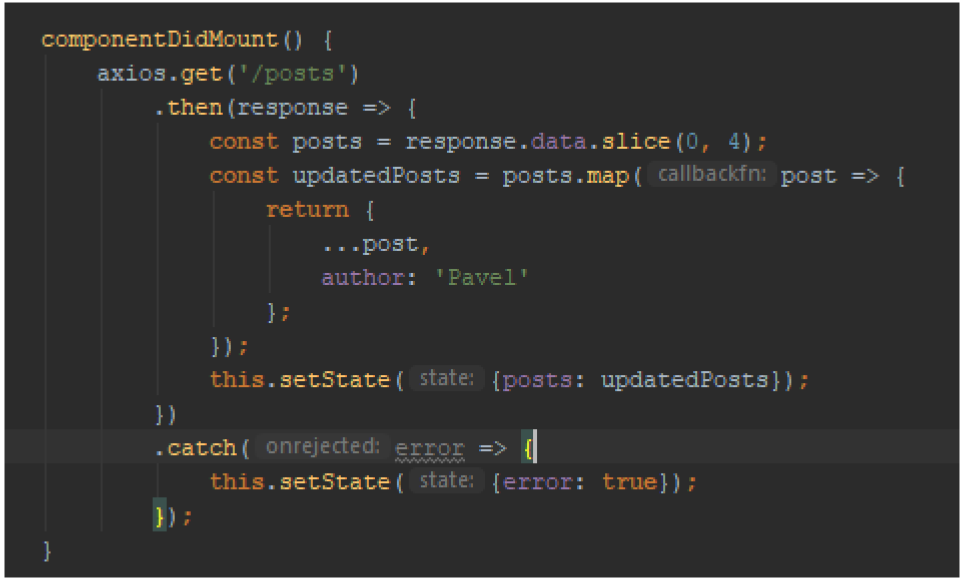


React HTTP

- Fake Online REST API for Testing and Prototyping:
  - http://jsonplaceholder.typicode.com/
- It is common to use third-party library -> Axios (preview the Readme.MD for more info):
  - https://github.com/axios/axios
  - Axios can be used in any JavaScript code
  - ‘npm install axios --save’

Where Should We Update The State? HTTP GET

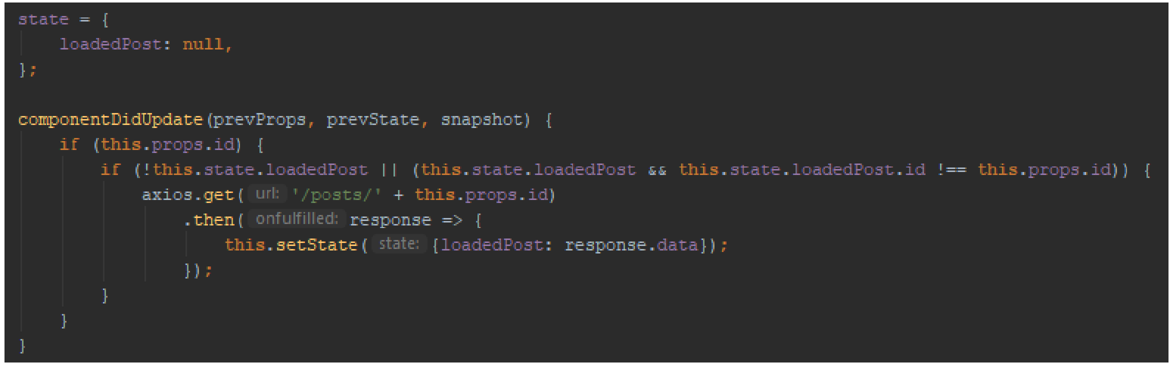
- Remember the predefined methods from React?
- Use componentDidMount to request data..
  - We cannot save data directly like const c1 = axios.get(...);
  - This is because it is invoked asynchronously
- Data are transformed using **map** method



```
componentDidMount() {
    axios.get('/posts')
        .then(response => {
            const posts = response.data.slice(0, 4);
            const updatedPosts = posts.map( callbackfn: post => {
                return {
                    ...post,
                    author: 'Pavel'
                };
            });
            this.setState( state: {posts: updatedPosts});
        })
        .catch( onrejected: error => {
            this.setState( state: {error: true});
        });
}
```

Avoid Infinite Loops

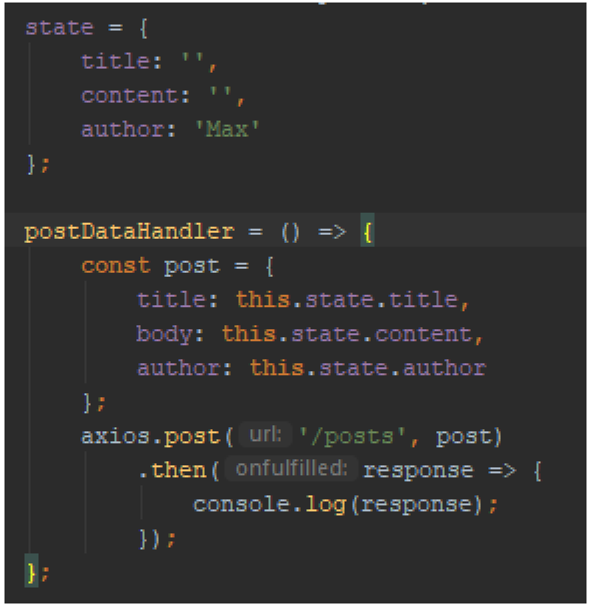
- Consider the method that we are getting a post based on its ID
- In practice, we need to check here if the loadedPost is valid to avoid loopingly re-rendering of the component
- Always make sure that you do not make infinite calls!!



```
state = {
    loadedPost: null,
};

componentDidUpdate(prevProps, prevState, snapshot) {
    if (this.props.id) {
        if (!this.state.loadedPost || (this.state.loadedPost && this.state.loadedPost.id !== this.props.id)) {
            axios.get( url: '/posts/' + this.props.id)
                .then( onfulfilled: response => {
                    this.setState( state: {loadedPost: response.data});
                });
        }
    }
}
```

HTTP POST

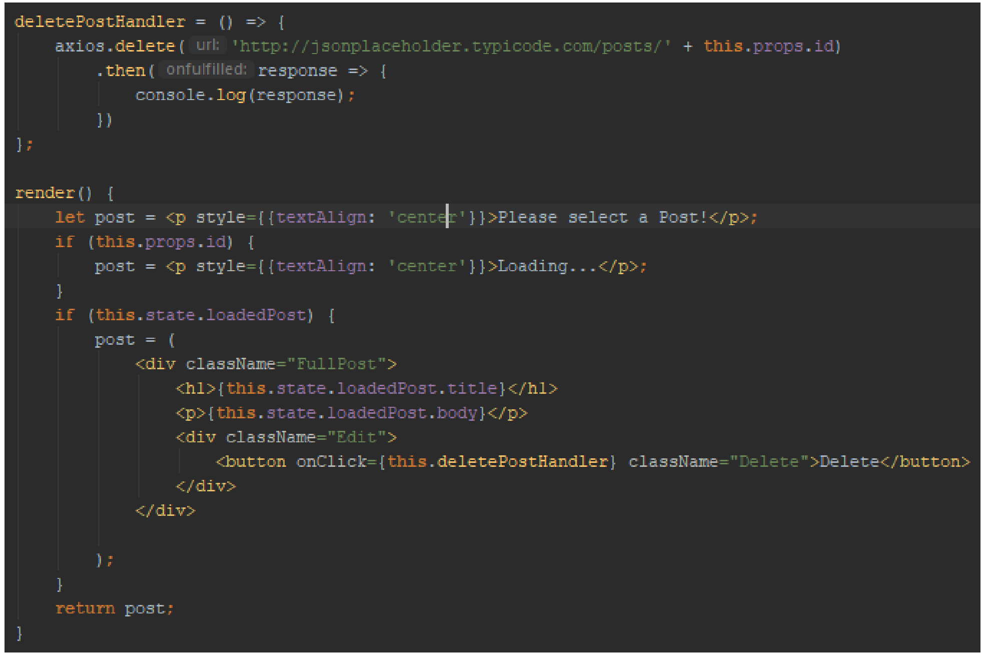
- Axios POST method requires a second argument to pass a data
- Basically, you just pass a values to the state of the component and use that state to send a data when a button is clicked



```
state = {
    title: '',
    content: '',
    author: 'Max'
};

postDataHandler = () => {
    const post = {
        title: this.state.title,
        body: this.state.content,
        author: this.state.author
    };
    axios.post( url: '/posts', post)
        .then( onfulfilled: response => {
            console.log(response);
        });
};
```

HTTP DELETE

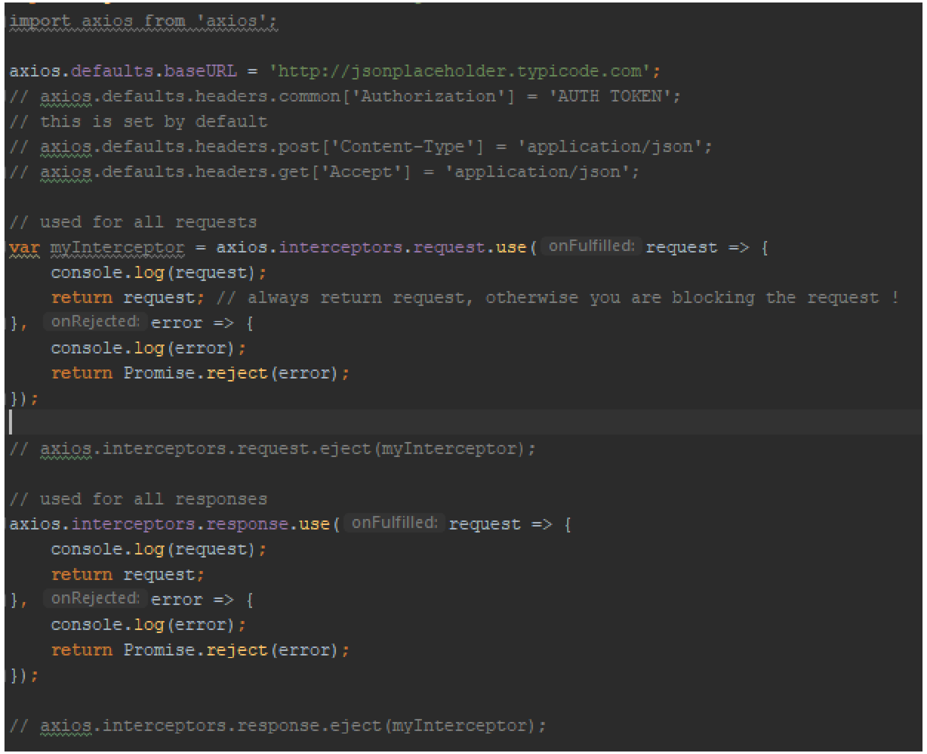
- Delete entity based on the ID
- Delete is idempotent operation
- Pay special attention that all the axios http methods are asynchronous



```
deletePostHandler = () => {
    axios.delete( url: 'http://jsonplaceholder.typicode.com/posts/' + this.props.id)
        .then( onfulfilled: response => {
            console.log(response);
        })
};

render() {
    let post = <p style={{textAlign: 'center'}}>Please select a Post!</p>;
    if (this.props.id) {
        post = <p style={{textAlign: 'center'}}>Loading...</p>;
    }
    if (this.state.loadedPost) {
        post = (
            <div className="FullPost">
                <h1>{this.state.loadedPost.title}</h1>
                <p>{this.state.loadedPost.body}</p>
                <div className="Edit">
                    <button onClick={this.deletePostHandler} className="Delete">Delete</button>
                </div>
            </div>

        );
    }
    return post;
}
```

React Interceptors

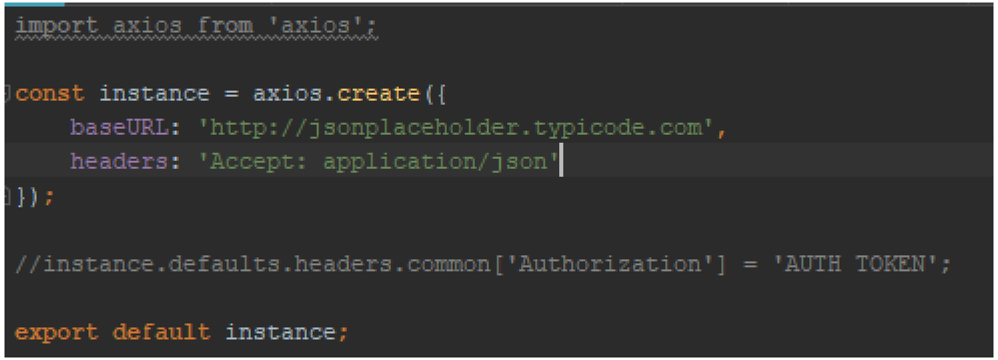
- These are functions that are defined globally that will be used in every request
- It is good to define them to add:
  - Error handling
  - Headers (Authorization, Accept, Content-Type, etc.)
  - Etc.
- Remove interceptors using **eject** method



```
import axios from 'axios';

axios.defaults.baseURL = 'http://jsonplaceholder.typicode.com';
// axios.defaults.headers.common['Authorization'] = 'AUTH TOKEN';
// this is set by default
// axios.defaults.headers.post['Content-Type'] = 'application/json';
// axios.defaults.headers.get['Accept'] = 'application/json';

// used for all requests
var myInterceptor = axios.interceptors.request.use( onFulfilled: request => {
    console.log(request);
    return request; // always return request, otherwise you are blocking the request !
}, onRejected: error => {
    console.log(error);
    return Promise.reject(error);
});

// axios.interceptors.request.eject(myInterceptor);

// used for all responses
axios.interceptors.response.use( onFulfilled: request => {
    console.log(request);
    return request;
}, onRejected: error => {
    console.log(error);
    return Promise.reject(error);
});

// axios.interceptors.response.eject(myInterceptor);
```

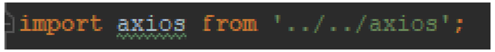
How to set several axios instances?

- Consider the scenario that we are querying several back-end services
  - We need to set different base urls, maybe, different content-types etc.
- How to achieve that?
- Luckily, axios provides so-called instances



```
import axios from 'axios';

const instance = axios.create({
    baseURL: 'http://jsonplaceholder.typicode.com',
    headers: 'Accept: application/json'
});

//instance.defaults.headers.common['Authorization'] = 'AUTH TOKEN';

export default instance;
```

- How to use that instance?
  - Just import axios instance that points to that file where you define your instance..
  - Instead of “import axios from `axios`”



```
import axios from '../../axios';
```

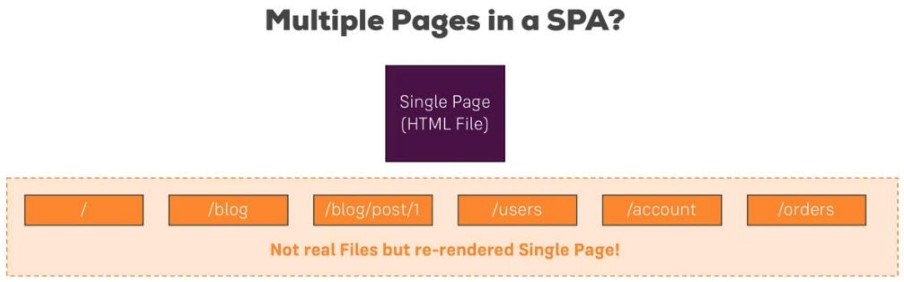
Tasks - Axios

1. Implement POST method in our predefined app {connect to back-end}
2. Implement DELETE method in our predefined app {connect to back-end}
3. Implement GET method in our predefined app {connect to back-end}
4. Implement two instances of axios
    a. Import the instance of axios that points to our back-end and use it to call the previously mentioned operations

Section: Routing

Routing

- Routing is about being able to show different pages to the user
- Commonly, the pages have pages like home, contact us, about, etc.
- How to provide Multiple Pages in a SPA
    - We actually, re-render the single page app
    - Basically, it parses URL /Path

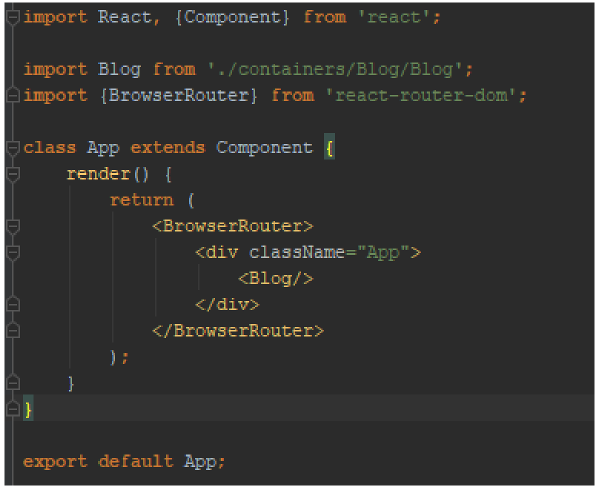


Configuration of React Routing

- **npm install --save react-router react-router-dom**
  - react-router -> exports the core components and functions
  - react-router-dom -> exports DOM-aware components, like <Link> (which renders an <a>) and <BrowserRouter> (which interacts with the browser's window.history)
  - In v4: **react-router-dom re-exports all of react-router's exports**, so you only need to import from **react-router-dom** in your project (it is not longer necessary to install react-router)
- These packages are not created by Facebook, but are de-facto standard for React routing

Start with React Routing

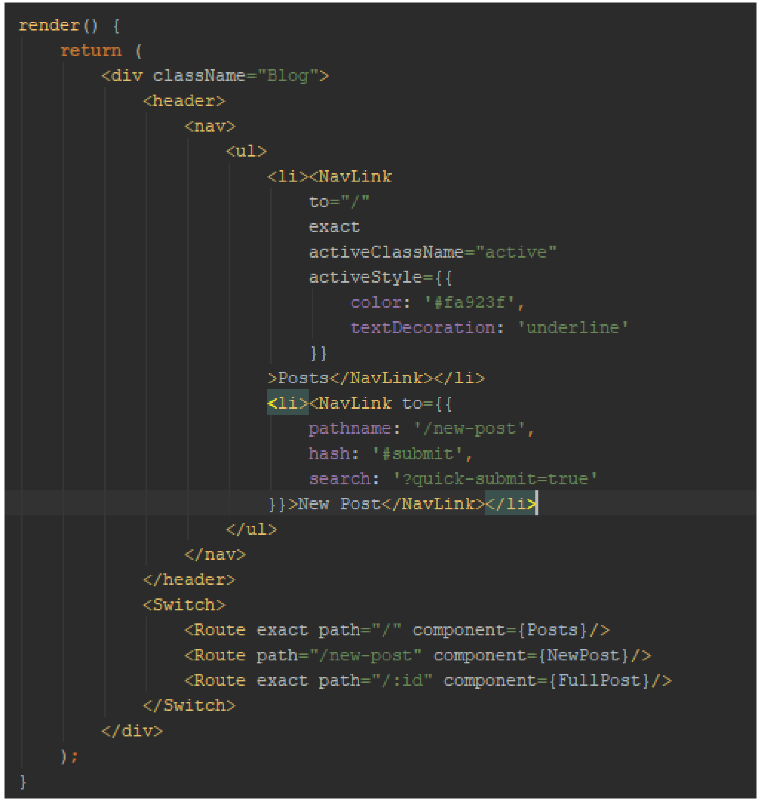
- Wrap the Index.js or App.js with BrowserRouter
  - Using this I can use routing in my child components
- A <Router> that uses the HTML5 history API (pushState, replaceState and the popstate event) to keep your UI in sync with the URL.
- History API:
  - https://css-tricks.com/using-the-html5-history-api/
  - “The HTML5 History API gives developers the ability to modify a website’s URL without a full page refresh. This is particularly useful for loading portions of a page with JavaScript, such that the content is significantly different and warrants a new URL. “



```
import React, {Component} from 'react';

import Blog from './containers/Blog/Blog';
import {BrowserRouter} from 'react-router-dom';

class App extends Component {
    render() {
        return (
            <BrowserRouter>
                <div className="App">
                    <Blog/>
                </div>
            </BrowserRouter>
        );
    }
}

export default App;
```

React Routing Used Tags

- Route
- Link
- NavLink (adds active class to the HTML)
- Switch



```
render() {
    return (
        <div className="Blog">
            <header>
                <nav>
                    <ul>
                        <li><NavLink
                            to="/"
                            exact
                            activeClassName="active"
                            activeStyle={{
                                color: '#fa923f',
                                textDecoration: 'underline'
                            }}
                        >Posts</NavLink></li>
                        <li><NavLink to={{
                            pathname: '/new-post',
                            hash: '#submit',
                            search: '?quick-submit=true'
                        }}>New Post</NavLink></li>
                    </ul>
                </nav>
            </header>
            <Switch>
                <Route exact path="/" component={Posts}/>
                <Route path="/new-post" component={NewPost}/>
                <Route exact path="/:id" component={FullPost}/>
            </Switch>
        </div>
    );
}
```

React Routing Used Tags

- Link
  - A **<Link>** component is used to navigate to an <indexentry content=" component:about"> existing route that is defined using the <Route> component.
- NavLink (adds active class to the HTML)
  - The **<NavLink>** component is similar to the <Link> component, except that several props can be specified that help you to conditionally add styling attributes to the rendered element.
- Route
  - The **<Route/>** component is one of the most important building blocks in the React Router package. It renders the appropriate user interface when the current location matches the route's path
- Switch
  - The **Switch** component **only** picks the **first** matching route among all its children routes.

Absolute vs Relative Paths

- Always generate absolute path

```
<Link to={{
 pathname: ‘/persons’,
 ...
}}
```

- If you need to append path to the current path use:

```
<Link to={{
 pathname: this.props.match.url + ‘/persons’,
 ...
}}
```

Passing Route Parameters

- **Route params** are parameters whose values are set dynamically in a page’s URL
- This allows a route to render the same component while passing that component the dynamic portion of the URL, so that it can change its data based on the parameter

```
<Route exact path=”/persons/:id” component={PersonsContainer} />
```

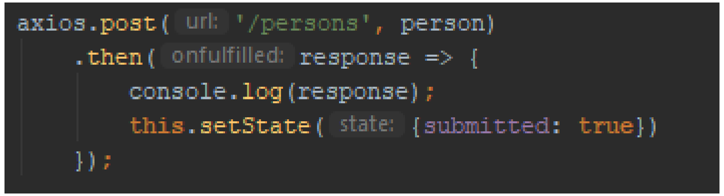
- The “/persons” part is static, but “/:id” part is generated automatically usually based on some click or something like that

Extracting Route Parameters

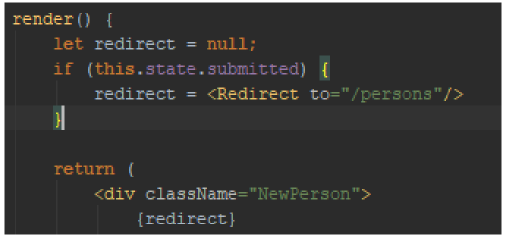
- The passed parameters can be retrieved using:
  - this.props.match.params.userId;
  - This.props.location.search
- Do not forget to add (“import {withRouter} from ‘react-router-dom’”) in this component and wrapp exported function into withRouter, e.g., as follows: “export default withRouter(PersonsContainer);”

Redirect

- Consider the scenario, when we want to redirect from “/” to “/home” always
  - We can use the <Redirect> tag from ‘react-router-dom’ as follows: `<Redirect from="/" to="/home"/>`
- Common scenario for redirection is after form submission:
  - Consider the scenario, when you submit a form and you want to redirect back to other page
  - 1. Add additional state property `submitted: false,`
  - 2. Set state after form submission:
  - 3. Conditionally redirect

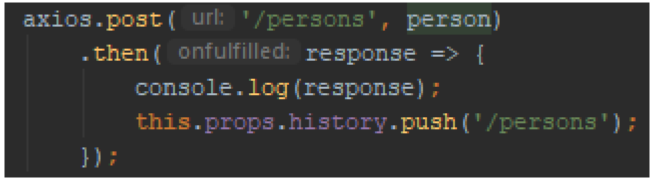


```
axios.post( url: '/persons', person)
    .then( onfulfilled: response => {
        console.log(response);
        this.setState( state: {submitted: true})
    });
```



```
render() {
    let redirect = null;
    if (this.state.submitted) {
        redirect = <Redirect to="/persons"/>
    }

    return (
        <div className="NewPerson">
            {redirect}
```

- Additional approach is to use HTML5 History:

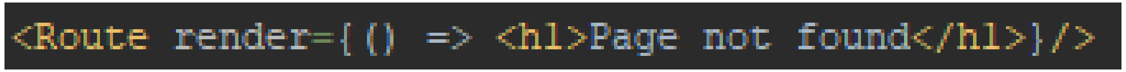


```
axios.post( url: '/persons', person)
    .then( onfulfilled: response => {
        console.log(response);
        this.props.history.push('/persons');
    });
```

  - The difference is that using Redirect you cannot go back since it replaces the page
    - (To achieve a same behaviour we can use “this.props.history.replace(‘/persons’));
    - However, <Redirect> tag is still useful in conditional redirection

Return 404 Page

- For any page that do not match the Routes add something as follows:



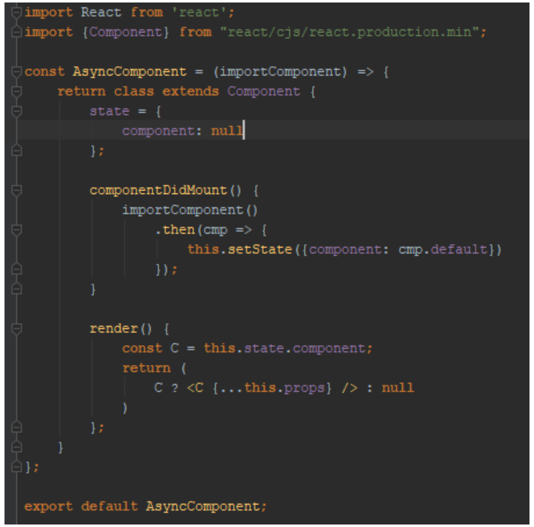
```
<Route render={() => <h1>Page not found</h1>}/>
```

- In the case, that none Route is matched, this Route renders that the page is not found (ofcourse, in practice, it will be some nicely formatted page)

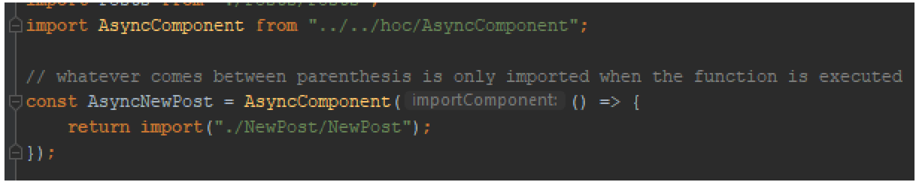
Loading Routes Lazily

- The goal: Download only code that is responsible for particular components
  - Useful in large applications
  - This technique is sometimes called “code splitting” or “lazy loading”

Proprietary HOC Component

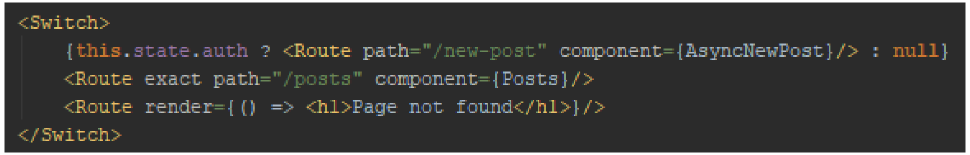


```
import React from 'react';
import {Component} from "react/cjs/react.production.min";

const AsyncComponent = (importComponent) => {
    return class extends Component {
        state = {
            component: null
        };

        componentDidMount() {
            importComponent()
                .then(cmp => {
                    this.setState({component: cmp.default})
                });
        }

        render() {
            const C = this.state.component;
            return (
                C ? <C {...this.props} /> : null
            )
        };
    }
};

export default AsyncComponent;
```



```
import AsyncComponent from "../../hoc/AsyncComponent";

// whatever comes between parenthesis is only imported when the function is executed
const AsyncNewPost = AsyncComponent( importComponent: () => {
    return import("./NewPost/NewPost");
});
```

Loaded only when the component is required

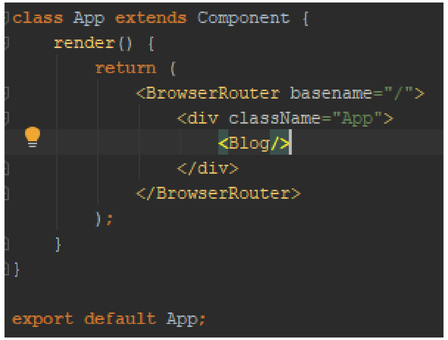


```
<Switch>
    {this.state.auth ? <Route path="/new-post" component={AsyncNewPost}/> : null}
    <Route exact path="/posts" component={Posts}/>
    <Route render={() => <h1>Page not found</h1>}/>
</Switch>
```

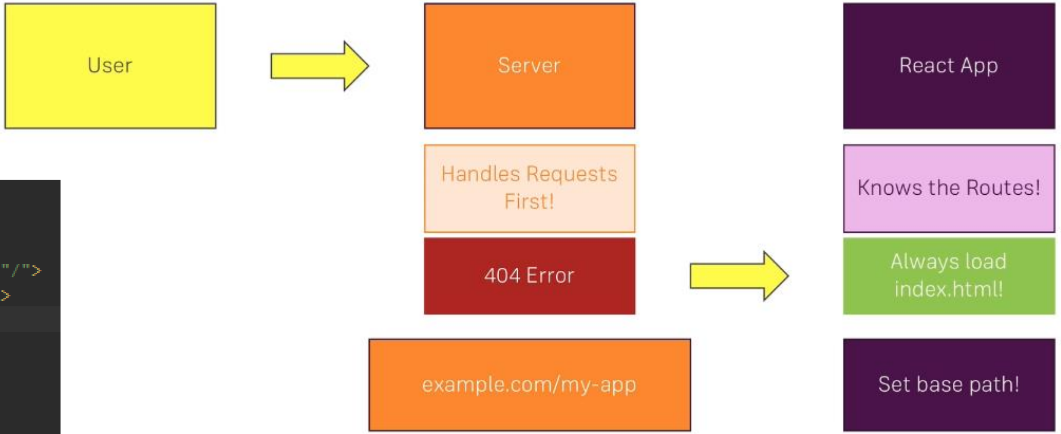
Note: Another alternative for Lazy Loading is to use React Suspense (React 16.6)

Routing & Server (Deployment)

- Always load index.html
    - Your server does not know anything about the routes



```
class App extends Component {
    render() {
        return (
            <BrowserRouter basename="/">
                <div className="App">
                    <Blog/>
                </div>
            </BrowserRouter>
        );
    }
}

export default App;
```



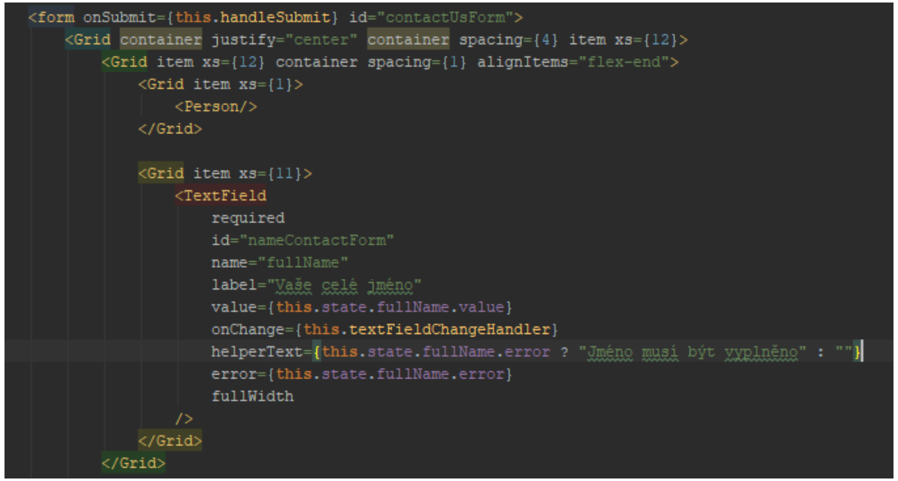
- It's better to add BrowserRouter to index.js
- Add basename: <protocol>://<IP-address>:<port>/<base-name>

Tasks - Implement a routing using <BrowserRouter>

1. Create a pages with the following header:
   a. Home (some basic info page), About us (load persons), Contact us (load addresses), …
      i. Use Route and Link/NavLink (check what happened when you add just <a href=""> instead of <Link …>
      ii. Consider exact paths for selected scenarios
2. Make the persons clickable by adding a link and load the “Person” component in the place of “Persons”
3. Pass the person ID to the “Person” page and output it there
   a. Optional: Pass additional params (select just one, e.g., username), try to use query params (you will need to parse them manually…)
4. Include a <Switch> when necessary
5. Add 404 error page and render it for unknown routes
6. Provide Redirect from “/” to “/home”

Section: Forms and Validation

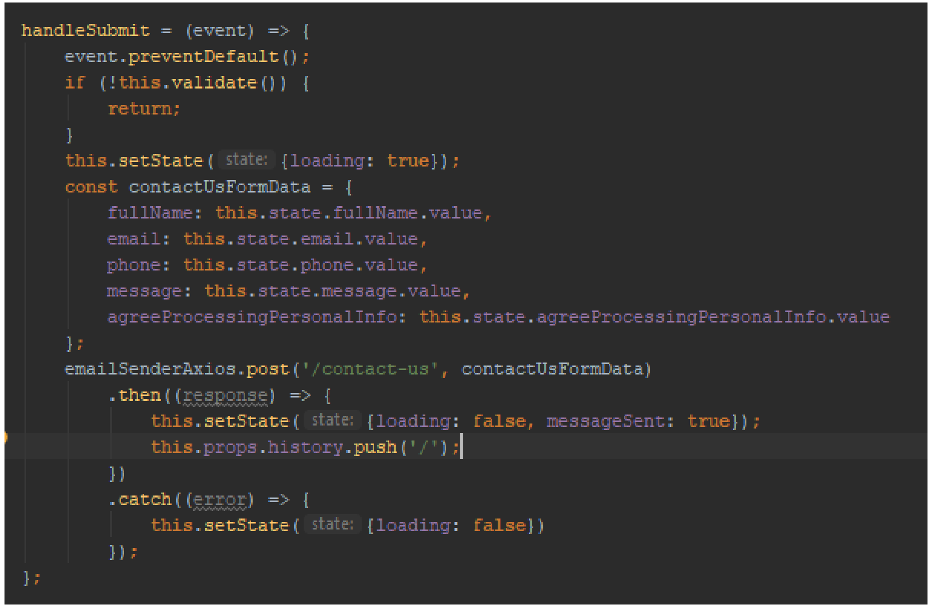
- The forms and form processing in React is quite standard
- Basically just add:
  - <form onSubmit={this.handleSubmit} …> input fields </form>



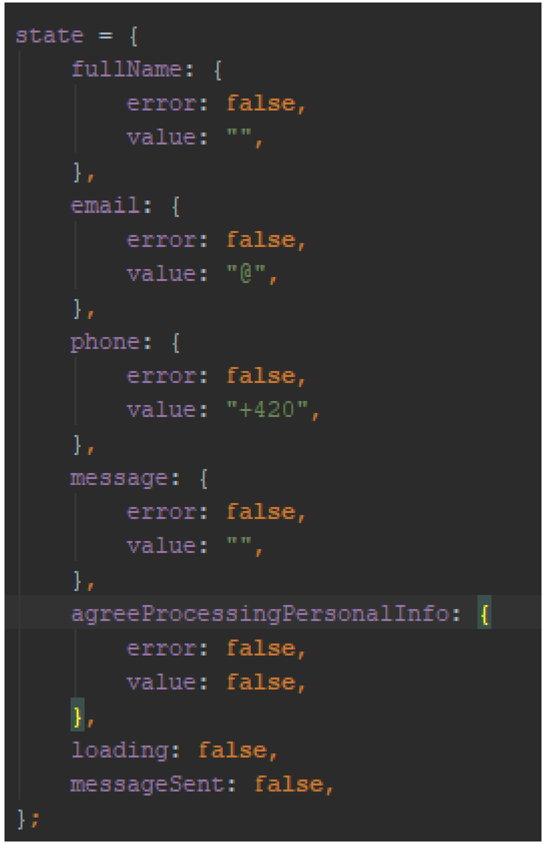
```
<form onSubmit={this.handleSubmit} id="contactUsForm">
    <Grid container justify="center" container spacing={4} item xs={12}>
        <Grid item xs={12} container spacing={1} alignItems="flex-end">
            <Grid item xs={1}>
                <Person/>
            </Grid>

            <Grid item xs={11}>
                <TextField
                    required
                    id="nameContactForm"
                    name="fullName"
                    label="Vaše celé jméno"
                    value={this.state.fullName.value}
                    onChange={this.textFieldChangeHandler}
                    helperText={this.state.fullName.error ? "Jméno musí být vyplněno" : ""}
                    error={this.state.fullName.error}
                    fullWidth
                />
            </Grid>
        </Grid>
```

Form

- Designing the State in the form submission
- Handle Submit -- and validation



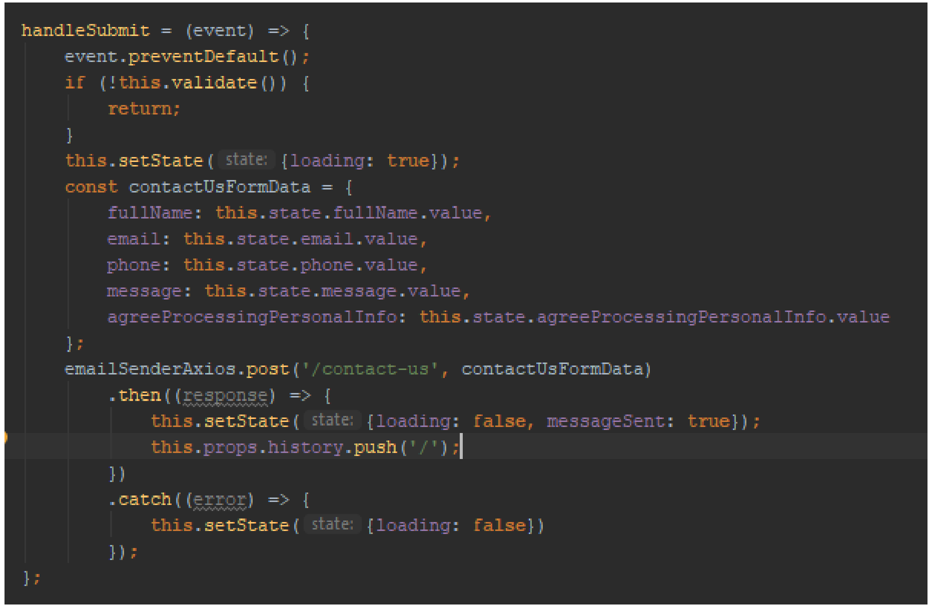
```
handleSubmit = (event) => {
    event.preventDefault();
    if (!this.validate()) {
        return;
    }
    this.setState( state: {loading: true});
    const contactUsFormData = {
        fullName: this.state.fullName.value,
        email: this.state.email.value,
        phone: this.state.phone.value,
        message: this.state.message.value,
        agreeProcessingPersonalInfo: this.state.agreeProcessingPersonalInfo.value
    };
    emailSenderAxios.post('/contact-us', contactUsFormData)
        .then((response) => {
            this.setState( state: {loading: false, messageSent: true});
            this.props.history.push('/');
        })
        .catch((error) => {
            this.setState( state: {loading: false})
        });
};
```



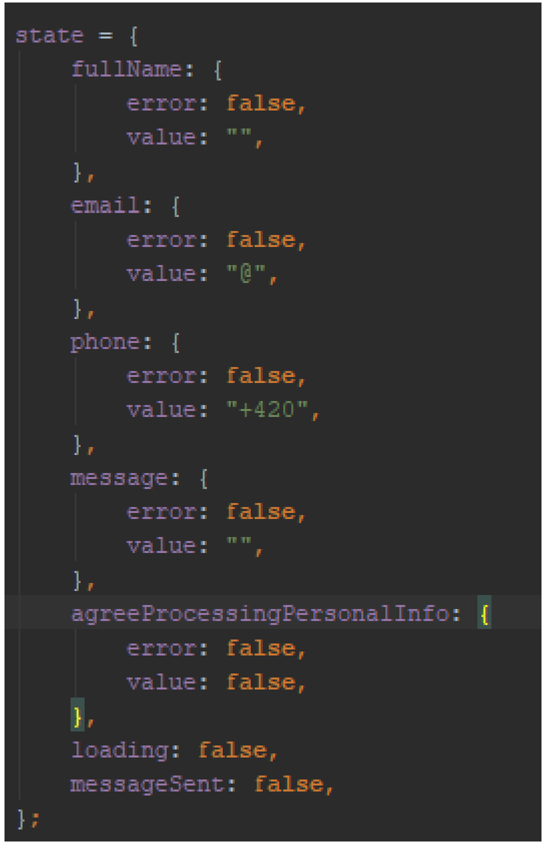
```
state = {
    fullName: {
        error: false,
        value: "",
    },
    email: {
        error: false,
        value: "@",
    },
    phone: {
        error: false,
        value: "+420",
    },
    message: {
        error: false,
        value: "",
    },
    agreeProcessingPersonalInfo: {
        error: false,
        value: false,
    },
    loading: false,
    messageSent: false,
};
```

Form

- Designing the State in the form submission
- Handle Submit -- and validation



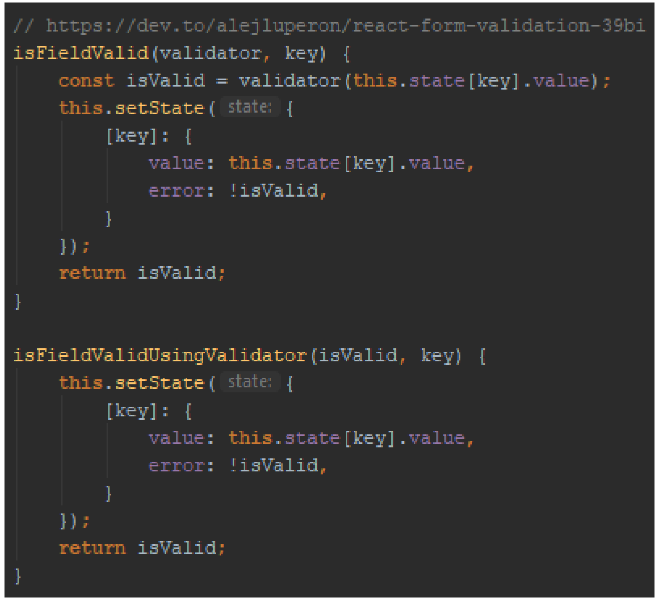
```
handleSubmit = (event) => {
    event.preventDefault();
    if (!this.validate()) {
        return;
    }
    this.setState( state: {loading: true});
    const contactUsFormData = {
        fullName: this.state.fullName.value,
        email: this.state.email.value,
        phone: this.state.phone.value,
        message: this.state.message.value,
        agreeProcessingPersonalInfo: this.state.agreeProcessingPersonalInfo.value
    };
    emailSenderAxios.post('/contact-us', contactUsFormData)
        .then((response) => {
            this.setState( state: {loading: false, messageSent: true});
            this.props.history.push('/');
        })
        .catch((error) => {
            this.setState( state: {loading: false})
        });
};
```



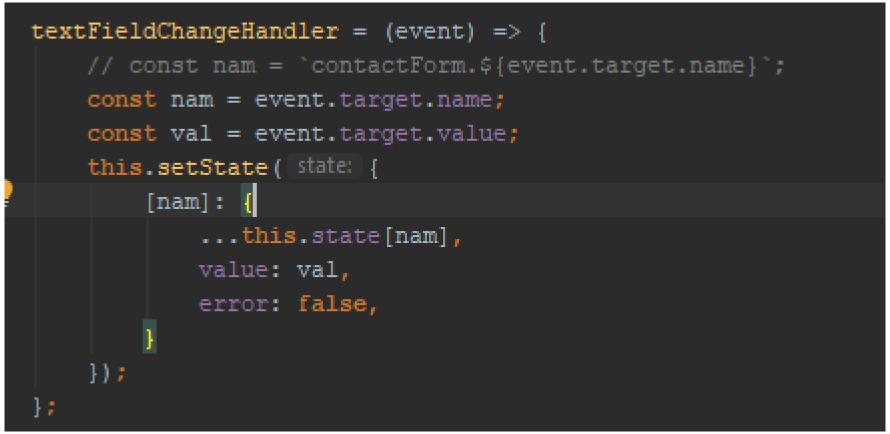
```
state = {
    fullName: {
        error: false,
        value: "",
    },
    email: {
        error: false,
        value: "@",
    },
    phone: {
        error: false,
        value: "+420",
    },
    message: {
        error: false,
        value: "",
    },
    agreeProcessingPersonalInfo: {
        error: false,
        value: false,
    },
    loading: false,
    messageSent: false,
};
```

Form Validation

- Install validator package and import the validator
  - import validator from "validator";



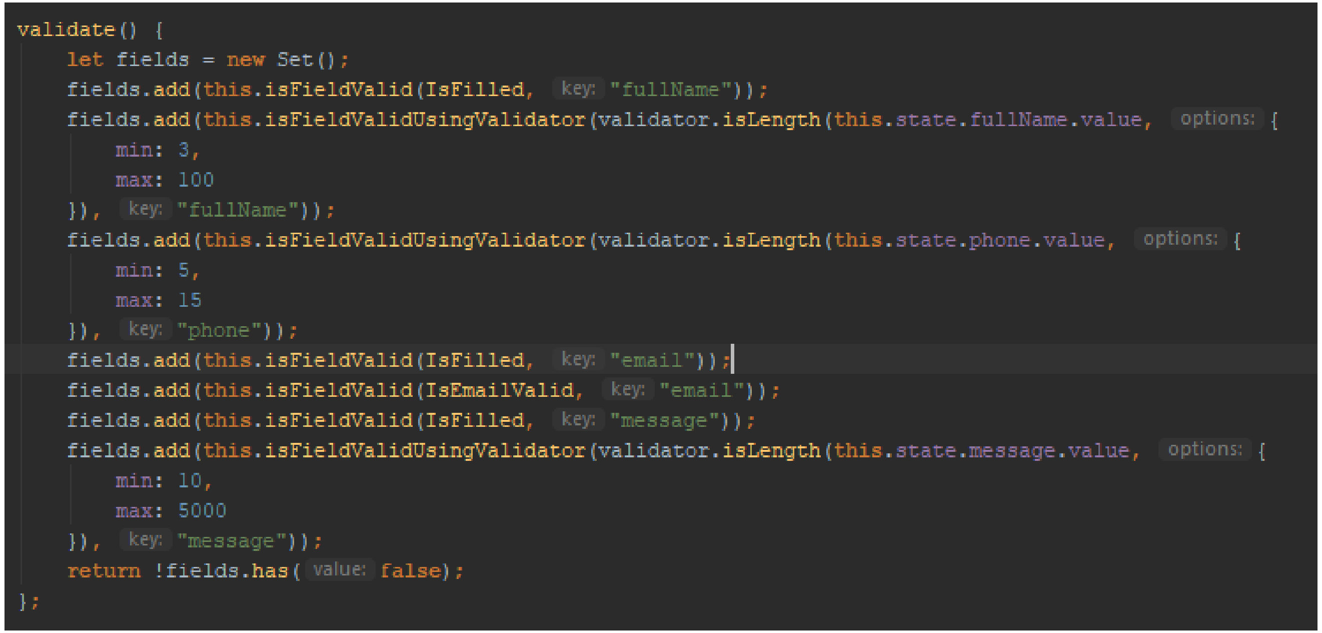
```
// https://dev.to/alejluperon/react-form-validation-39bi
isFieldValid(validator, key) {
    const isValid = validator(this.state[key].value);
    this.setState( state: {
        [key]: {
            value: this.state[key].value,
            error: !isValid,
        }
    });
    return isValid;
}

isFieldValidUsingValidator(isValid, key) {
    this.setState( state: {
        [key]: {
            value: this.state[key].value,
            error: !isValid,
        }
    });
    return isValid;
}
```



```
textFieldChangeHandler = (event) => {
    // const nam = `contactForm.${event.target.name}`;
    const nam = event.target.name;
    const val = event.target.value;
    this.setState( state: {
        [nam]: {
            ...this.state[nam],
            value: val,
            error: false,
        }
    });
};
```

Form Validation

- Before form submission validate the input fields



```
validate() {
    let fields = new Set();
    fields.add(this.isFieldValid(IsFilled, key: "fullName"));
    fields.add(this.isFieldValidUsingValidator(validator.isLength(this.state.fullName.value, options: {
        min: 3,
        max: 100
    }), key: "fullName"));
    fields.add(this.isFieldValidUsingValidator(validator.isLength(this.state.phone.value, options: {
        min: 5,
        max: 15
    }), key: "phone"));
    fields.add(this.isFieldValid(IsFilled, key: "email"));
    fields.add(this.isFieldValid(IsEmailValid, key: "email"));
    fields.add(this.isFieldValid(IsFilled, key: "message"));
    fields.add(this.isFieldValidUsingValidator(validator.isLength(this.state.message.value, options: {
        min: 10,
        max: 5000
    }), key: "message"));
    return !fields.has( value: false);
};
```

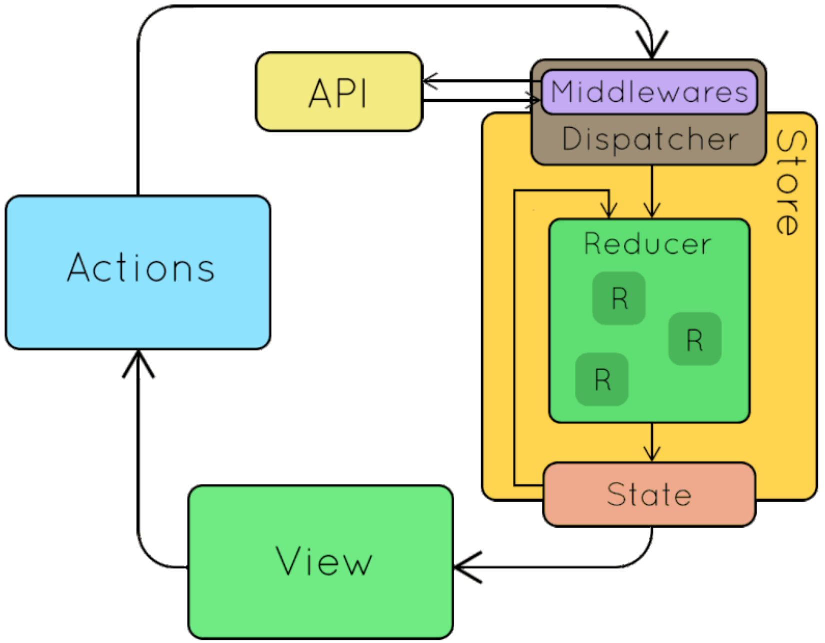
Section: Redux

Redux

- What is redux (https://redux.js.org/)?
    - A stand-alone library used to simplify state management in JavaScript projects
- Why Redux?
    - Commonly used to React to easier state management
    - In simple words, it treats like a central state management store where we look for the current state
    - Sometimes it is difficult to manage a state between totally different components (e.g., auth…)
- With what Redux can help?
    - Detect if user is authenticated (what to render if auth or not)
    - Checking if Modal window is open
    - ...
- Dan Abramov, one of the creators of Redux, says:
    - “I would like to amend this: don't use Redux until you have problems with vanilla React.”

How Redux Works?

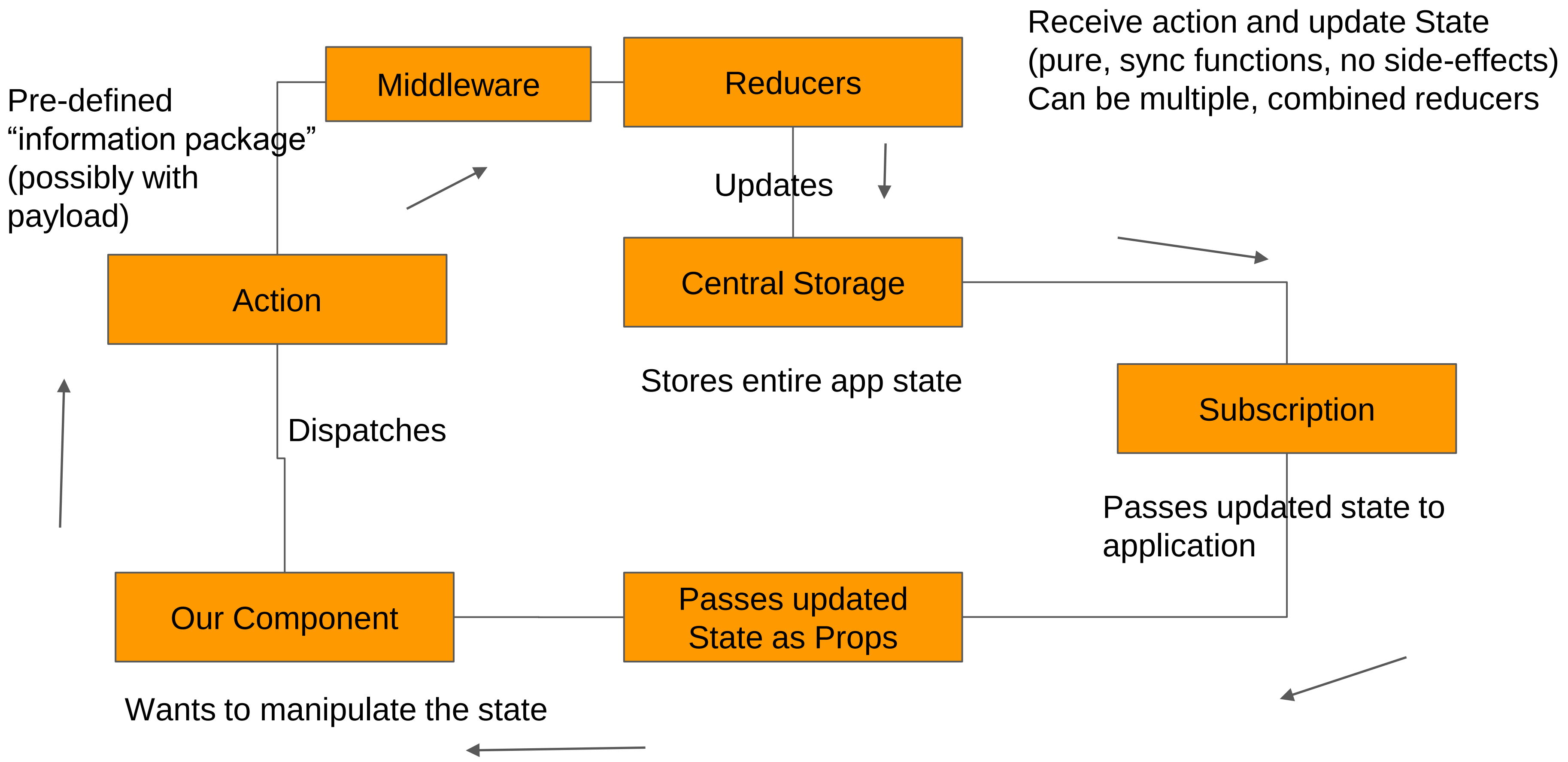
- Actions
  - Simply put, actions are events
  - User interactions, API calls, ...
- Store
  - Holds the application state
  - There is only one store in any Redux
- Reducer
  - Pure function that take the current state of an app, perform an action, and return a new state
- Dispatcher
  - Triggers a state change



Reducer and Store in React

- Installing Redux in React app
  - npm install -g redux react-redux
  - Check if dependency pears should not be installed also

Middleware

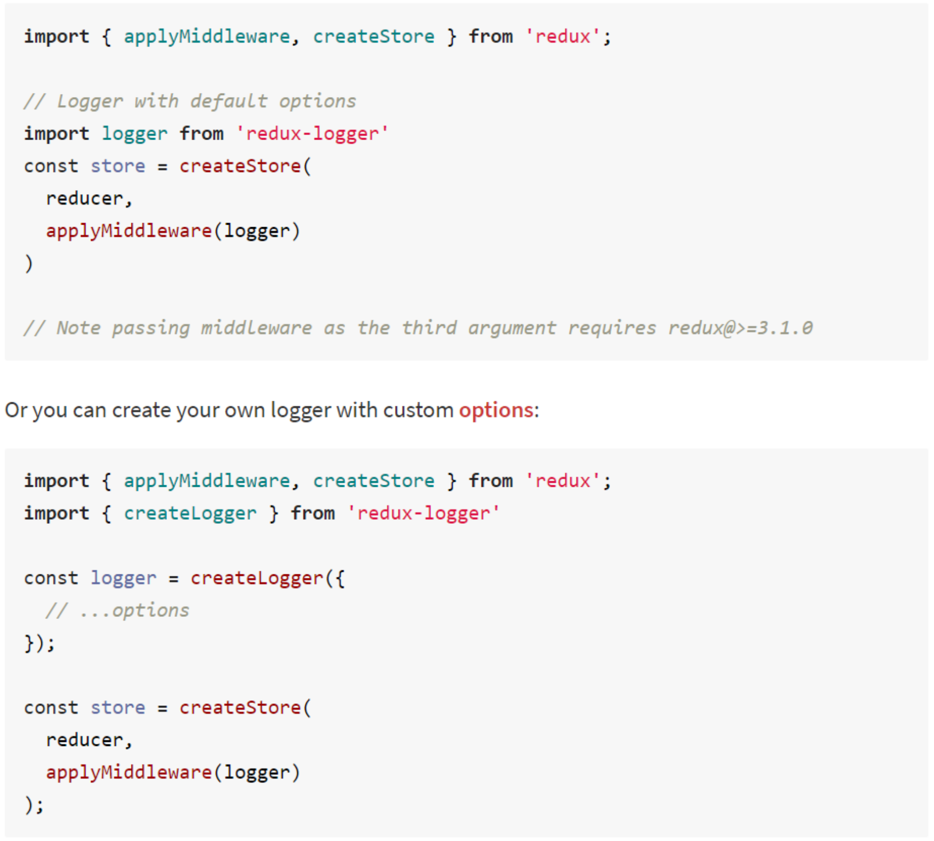

Pre-defined
“information package”
(possibly with
payload)
Middleware
Reducers
Receive action and update State
(pure, sync functions, no side-effects)
Can be multiple, combined reducers
Updates
Action
Central Storage
Stores entire app state
Subscription
Dispatches
Passes updated state to
application
Our Component
Passes updated
State as Props
Wants to manipulate the state

Middleware

- It provides a third-party extension point between dispatching an action, and the moment it reaches the reducer
- People use Redux middleware for:
  - Logging,
  - Crash reporting,
  - Talking to an asynchronous API,
  - Routing, and more

Middleware - Logging

- Consider 'redux-logger'
  - https://www.npmjs.com/package/redux-logger
- You can configure logger to, e.g., be invoked only in development mode or just to be ignored if a special type of action is invoked etc.



```
import { applyMiddleware, createStore } from 'redux';

// Logger with default options
import logger from 'redux-logger'
const store = createStore(
  reducer,
  applyMiddleware(logger)
)

// Note passing middleware as the third argument requires redux@>=3.1.0
```

Or you can create your own logger with custom options:

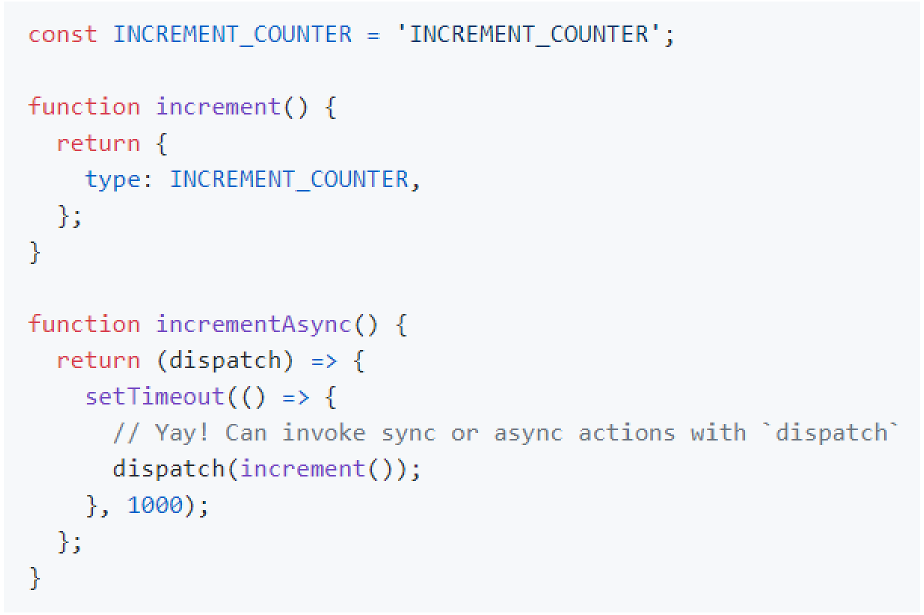
```
import { applyMiddleware, createStore } from 'redux';
import { createLogger } from 'redux-logger'

const logger = createLogger({
  // ...options
});

const store = createStore(
  reducer,
  applyMiddleware(logger)
);
```

Middleware - Thunk

- With a plain basic Redux store, you can only do simple synchronous updates by dispatching an action
- Middleware extends the store's abilities, and lets you write async logic that interacts with the store
- Thunks are the recommended middleware for basic Redux side effects logic, including complex synchronous logic that needs access to the store, and simple async logic like AJAX requests.
    - https://github.com/reduxjs/redux-thunk

Middleware - Thunk - Motivation

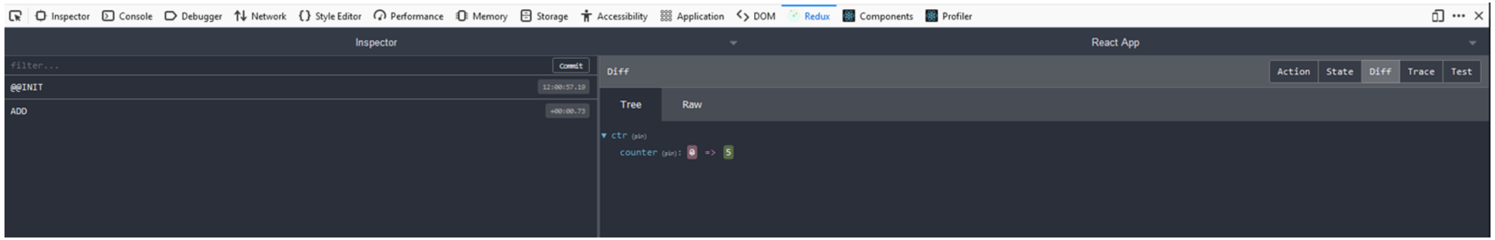
- Redux Thunk middleware allows you to write action creators that return a function instead of an action
- The thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met
- The inner function receives the store methods dispatch and getState as parameters.



```
const INCREMENT_COUNTER = 'INCREMENT_COUNTER';

function increment() {
  return {
    type: INCREMENT_COUNTER,
  };
}

function incrementAsync() {
  return (dispatch) => {
    setTimeout(() => {
      // Yay! Can invoke sync or async actions with `dispatch`
      dispatch(increment());
    }, 1000);
  };
}
```

Redux DevTools

- Redux DevTools for debugging application's state changes
- It's an open source project
- It can be used as a browser extension (for Chrome, Edge and Firefox), as a standalone app or as a React component integrated in the client app.
    - https://github.com/zalmoxisus/redux-devtools-extension
    - https://addons.mozilla.org/cs/firefox/addon/reduxdevtools
- Extremely useful tool



```
filter...
@@INIT    12:00:57.19
ADD       +00:00.73
```

```
▼ ctr (pin)
  counter (pin): 0 => 5
```

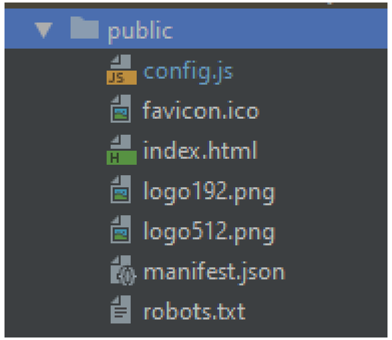
Redux Persisting the State

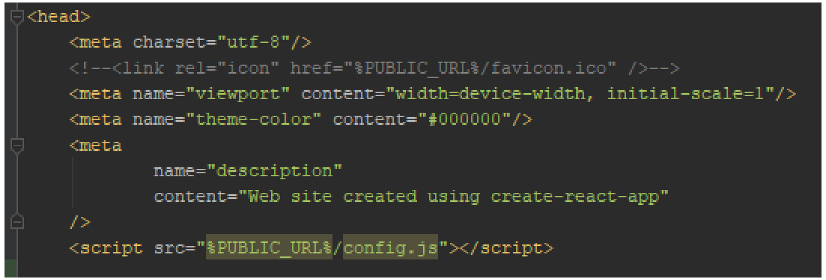
- If we do not use some kind of local storage our global state in Redux is removed..
- To persist the store you can consider the following:
  - https://www.npmjs.com/package/redux-persist

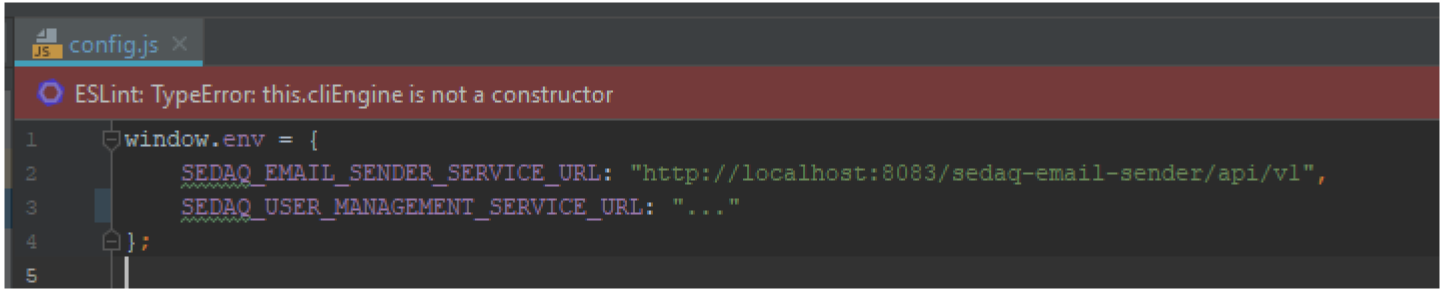
Section: React App Deployment

Consider external configuration for settings

- Add to the public folder `config.js` file (name it as you want)
- Add to index.html the reference to that config file as follows:


```
▼ public
    config.js
    favicon.ico
    index.html
    logo192.png
    logo512.png
    manifest.json
    robots.txt
```


```html
<head>
    <meta charset="utf-8"/>
    <!--<link rel="icon" href="%PUBLIC_URL%/favicon.ico" />-->
    <meta name="viewport" content="width=device-width, initial-scale=1"/>
    <meta name="theme-color" content="#000000"/>
    <meta
            name="description"
            content="Web site created using create-react-app"
    />
    <script src="%PUBLIC_URL%/config.js"></script>
```

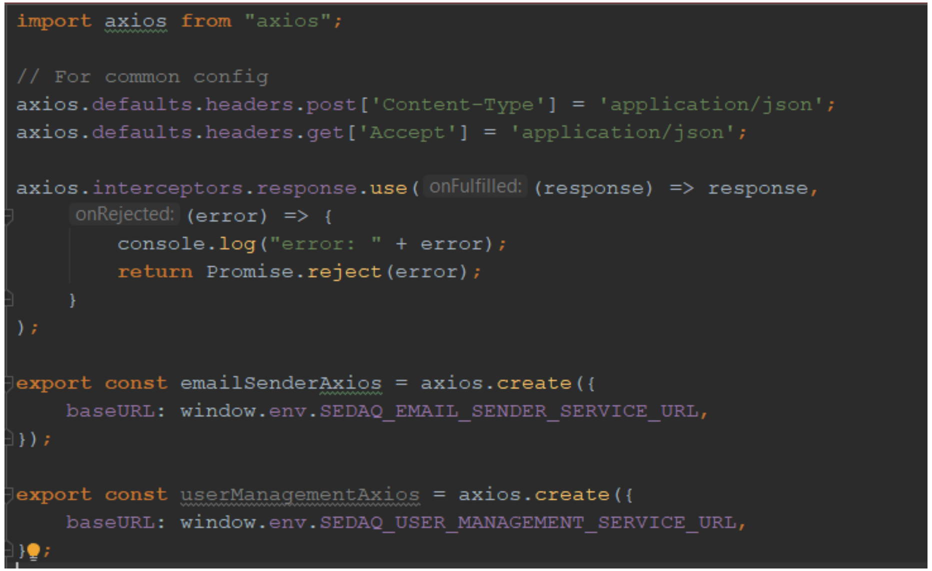

config.js

ESLint: TypeError: this.cliEngine is not a constructor

```js
window.env = {
    SEDAQ_EMAIL_SENDER_SERVICE_URL: "http://localhost:8083/sedaq-email-sender/api/v1",
    SEDAQ_USER_MANAGEMENT_SERVICE_URL: "..."
};

```

Use this configs in, e.g., Axios instances

- Reference that configs as:
    - window.env.XYZ



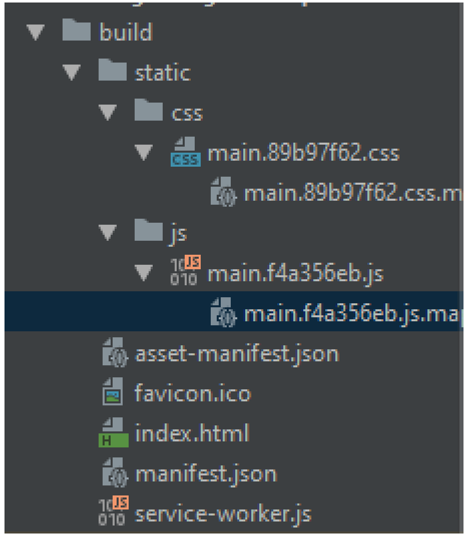
```javascript
import axios from "axios";

// For common config
axios.defaults.headers.post['Content-Type'] = 'application/json';
axios.defaults.headers.get['Accept'] = 'application/json';

axios.interceptors.response.use( onFulfilled: (response) => response,
     onRejected: (error) => {
        console.log("error: " + error);
        return Promise.reject(error);
    }
);

export const emailSenderAxios = axios.create({
    baseURL: window.env.SEDAQ_EMAIL_SENDER_SERVICE_URL,
});

export const userManagementAxios = axios.create({
    baseURL: window.env.SEDAQ_USER_MANAGEMENT_SERVICE_URL,
});
```

Deployment Steps

- Check and adjust basepath
  - <BrowserRouter basename=”/my-app”>
- Build & optimize the project
  - npm run build   (in create-react-app-project)
- Server must always serve index.html (especially for HTTP 404 cases)
  - Otherwise routing would not work correctly
- Upload build artifacts to (static) server
  - Upload /build artifacts (/builder folder)

Build the Project

- Just run “npm run build”
  - It will create a build folder in the project:



```
▼ build
    ▼ static
        ▼ css
            ▼ main.89b97f62.css
                  main.89b97f62.css.m
        ▼ js
            ▼ main.f4a356eb.js
                  main.f4a356eb.js.ma
      asset-manifest.json
      favicon.ico
      index.html
      manifest.json
      service-worker.js
```

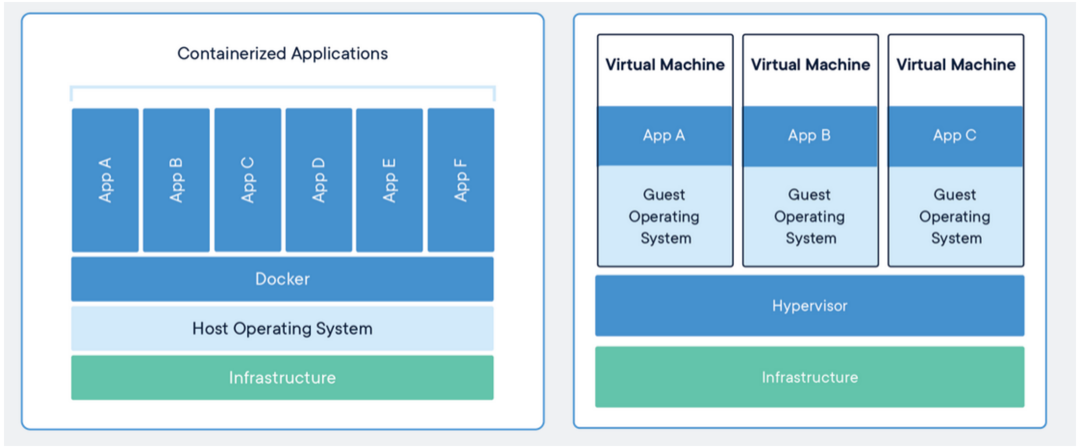
Section: Deployment with Docker

Docker

- Docker container technology was launched in 2013 as an open source Docker Engine
- Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers
- Containers are isolated from one another and bundle their own software, libraries and configuration files;
    - They can communicate with each other through well-defined channels
- All containers are run by a single operating system kernel and therefore use fewer resources than virtual machines
- Written in Go

Docker - Containers vs Virtual Machines

- Containers and virtual machines have similar resource isolation and allocation benefits, but function differently because **containers virtualize the operating system** instead of hardware
  - Containers are more portable and efficient



Docker - Terminology

- A **Docker image** is an immutable (unchangeable) file that contains the source code, libraries, dependencies, tools, and other files needed for an application to run
  - Due to their read-only quality, these images are sometimes referred to as snapshots
  - They represent an application and its virtual environment at a specific point in time
- A **Docker container** is a virtualized run-time environment where users can isolate applications from the underlying system
  - These containers are compact, portable units in which you can start up an application quickly and easily.

Docker - Important Files

- .Dockerfile
  - Provides instructions to create a Docker image
  - It contains commands such as:
    - FROM - creates a layer from the specific Docker image
    - COPY - adds files from layers or Docker client’s directory
    - RUN - executes any command in a new layer on top of the current image
    - CMD - used to run the software contained in your image, along with any arguments
    - EXPOSE - indicates the ports on which a container listens for connections
    - ENV - sets environment variables
    - ENTRYPOINT - allows us to configure a container that will run as an executable
- .dockerignore
  - The .dockerignore file allows you to exclude files from the context like a .gitignore file allow you to exclude files from your git repository
  - It helps to make build faster and lighter by excluding from the context big files or repository that are not used in the build

Docker - Best Practices

- Build the smallest image possible! (slim, alpine base images)
- Package a single app per container
- Persisting application data
  - Use data volumes
  - Use secrets to store sensitive application data used by services
- Use CI/CD for testing and deployment
  - For example, use Gitlab CI for that purpose

Docker - React Deployment Example .Dockerfile

```
FROM node:12 AS build-stage
WORKDIR /app
COPY package*.json /app/
RUN npm install
COPY ./ /app/
RUN npm run build

# Nginx stage
FROM nginx:alpine
COPY --from=build-stage /app/build/ /usr/share/nginx/html
COPY --from=build-stage /app/etc/nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 8000
CMD ["nginx", "-g", "daemon off;"]
```

Docker - .dockerignore example

```
node_modules
npm-debug.log
build
.dockerignore
**/.git
**/.DS_Store
**/node_modules
```

Docker - Nginx Configuration Example

- Here, we need to redirect 404 pages (not found), and always return index.html

```
server {
    listen 8000;

    root /usr/share/nginx/html;
                index index.html index.htm;

    # Any route that doesn't have a file extension (e.g. /contacts)
    location / {
                 try_files $uri $uri/ /index.html;
    }

}
```

Docker - Docker-compose Configuration Example

- The volumes and other runtime configurations can be passed from the Docker platform
- Basically, we can use Docker-compose (usually for development), and Docker Swarm or Kubernetes for production-ready deployments

```
version: '3.8'

services:
  gopas-frontend:
                  image: gopas/gopas-frontend
                  container_name: gopas-frontend
                  volumes:
                  - ./configuration/nginx.conf:/etc/nginx/conf.d/default.conf
                  networks:
                  - gopas-platform-net
                  ports:
                  - 80:80
  ...other back-end services…
  nginx-reverse-proxy:
          ….
networks:
  gopas-platform-net:
                  name: gopas-platform-net
                  driver: bridge
                  driver_opts:
                  com.docker.network.driver.mtu: 1500
```

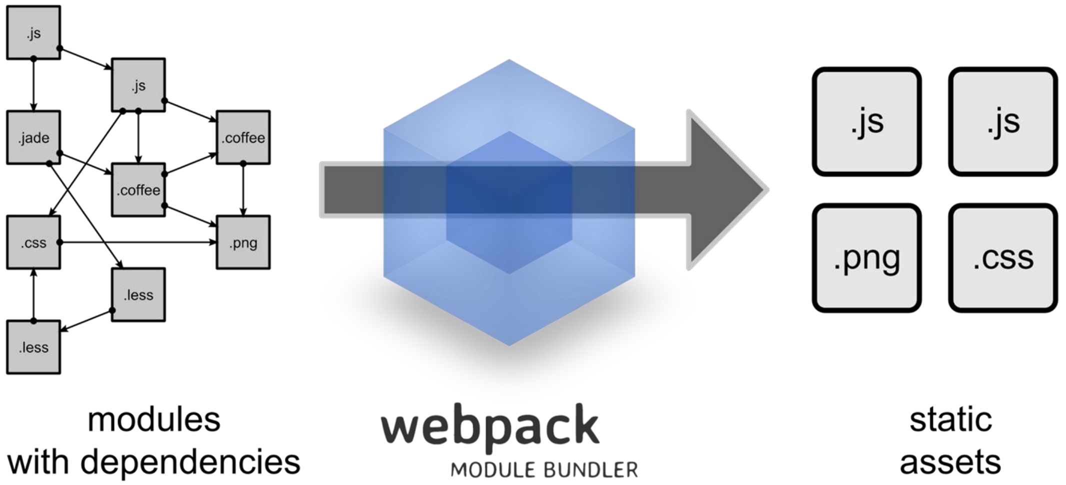
Docker - Build and Run Commands Example

- To build image run in the directory with Dockerfile the following command:
  - $ docker build -t gopas/gopas-frontend .
  - This will build the Docker image based on the instructions presented in Dockerfile
- Further, when the Docker image is builded run:
  - $ docker-compose up
- Open the localhost page
  - Now, the app should be successfully running

Section: Build Under the Hoods - Webpack

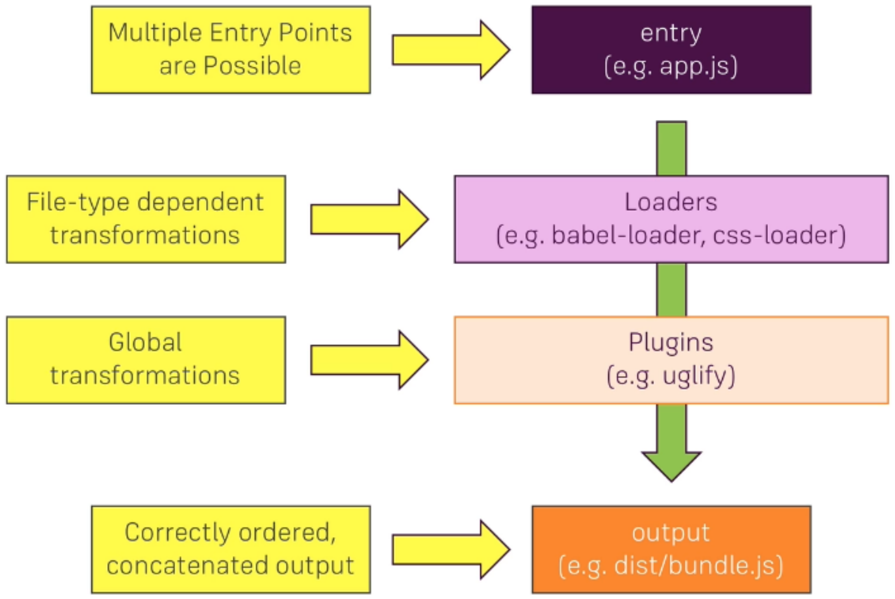
Webpack

- Webpack is a module bundler
- It takes modules with dependencies and generates static assets representing those modules

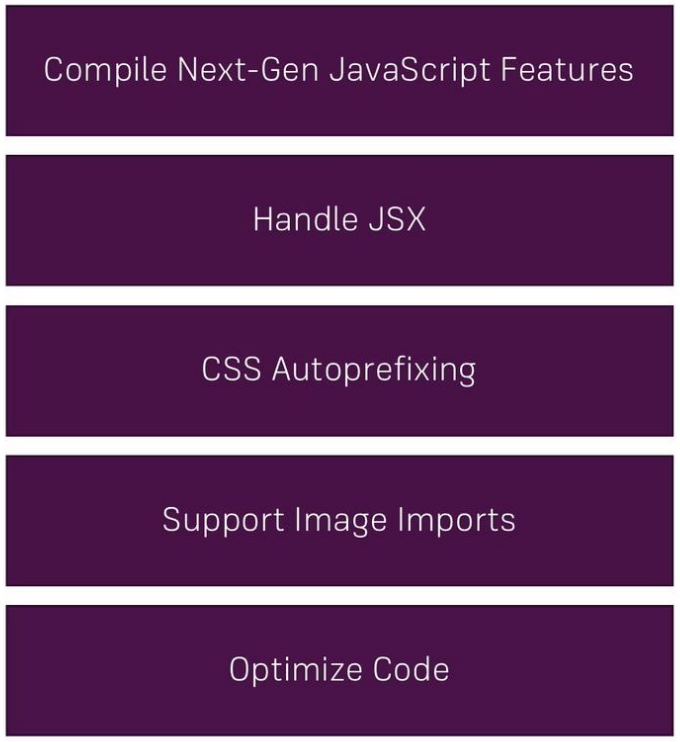


Webpack - How it Works?

- Webpack create dependency graph, starting with the entry file (e.g., app.js)


Multiple Entry Points are Possible
entry (e.g. app.js)
File-type dependent transformations
Loaders (e.g. babel-loader, css-loader)
Global transformations
Plugins (e.g. uglify)
Correctly ordered, concatenated output
output (e.g. dist/bundle.js)

Webpack - Basic Workflow Requirements



- Compile Next-Gen JavaScript Features
- Handle JSX
- CSS Autoprefixing
- Support Image Imports
- Optimize Code

What we did not cover

- Redux
- React Hooks in detail
- The possibility to include TypeScript into React